IOWA STATE UNIVERSITY
Digital Repository

Retrospective Theses and Dissertations

Iowa State University Capstones, Theses and Dissertations

1989

Toward the development and implementation of object-oriented extensions for discrete-event simulation in a strongly-typed procedural language

Kurt Hollister Diesch
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

 Part of the Industrial Engineering Commons

Recommended Citation
Diesch, Kurt Hollister, "Toward the development and implementation of object-oriented extensions for discrete-event simulation in a strongly-typed procedural language " (1989). *Retrospective Theses and Dissertations*. 8927.
https://lib.dr.iastate.edu/rtd/8927

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book. These are also available as one exposure on a standard 35mm slide or as a 17" x 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 8920124

Toward the development and implementation of object-oriented extensions for discrete-event simulation in a strongly-typed procedural language

Diesch, Kurt Hollister, Ph.D.

Iowa State University, 1989

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

Toward the development and implementation of object-oriented extensions for discrete-event simulation in a strongly-typed procedural language

by

Kurt Hollister Diesch

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Industrial Engineering

Approved:

Signature was redacted for privacy.

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Members of the Committee:

Signature was redacted for privacy.

Iowa State University
Ames, Iowa

1989

TABLE OF CONTENTS

I. INTRODUCTION

A. Preamble

The primary emphasis of this research is computer simulation. Computer simulations are used to model and analyze systems. To date, computer simulations have almost exclusively been written in procedural, strongly-typed languages such as FORTRAN or Pascal.

Recent advancements in simulation research suggest an object-oriented approach to simulation languages may provide key benefits in computer simulation. The goal of this research is to combine the advantages of a simulation language written in a procedural, strongly-typed language with the benefits available through the object-oriented programming paradigm.

The software developed in this research is capable of simulating systems with multiple servers and queues. Arrival and service distributions may be selected from the uniform, exponential, and normal family of distributions. Resource usage is not supported in the simulation program.

B. Statement of the Problem

Computer simulation can closely represent the real time behavior of systems while concurrently reducing the costs associated with data collection and study of real world systems. Simulation languages such as GPSS, SLAM, and many others have contributed significantly to simulation capabilities.

Most of the currently available simulation languages are based on strongly-typed traditional programming languages such as FORTRAN. Simulation models using these standard programming languages are typically constructed in

much the same fashion as most computer programs. The user must generate line after line of complicated computer code. The simulation modeler must generally be qualified as a simulation expert as well as a computer programmer. Details that can not be handled by the standard constructs of the language are added as (typically) FORTRAN inserts to the language.

Many of the manufacturers of simulation languages have recently recognized that building simulation models using standard programming languages is complex and results are often difficult to analyze. Few end users choose to invest the time and money required to generate even the simplest of simulation models. The complexity of simulation restricts the use of many languages to a minority of highly trained experts. The first natural extension to the original simulation languages was to add graphic or menu interfaces to the language in an effort to remove the programming complexity from model generation.

The results of this effort toward reduced complexity are twofold. While menus and graphic interfaces have effectively reduced operational complexity of the programs, versatility has suffered. Some manufacturers of simulation software choose to exclude all programming from their languages, but can not incorporate all possible simulation model requirements into their menus or graphic interfaces. The result is a language that is not capable of adequately modeling complex systems.

Other simulation software manufacturers retain the option of including external FORTRAN (or other language) inserts into the simulation model, but the effect is to allow the end user to use the menus or graphic interfaces for the portion of the modeling task that is already easy. The user must still perform the complex programming tasks for the difficult portions of the simulation model.

Another major problem with the current base of simulation languages is in the underlying language itself. Computer languages such as FORTRAN, BASIC, Pascal, C, and others, are all based on a concept known as sequential processing. Commands contained in the computer code must be processed one by one in a sequential fashion. Real-world systems, on the other hand, operate in a multi-process environment, where many activities occur simultaneously. In an effort to model real-world systems, current simulation languages utilize a built-in clock that is incremented by the software after all activities scheduled for a particular time have been completed. Only the increased processing power of computers has allowed simulation to mimic real-world systems with acceptable speed. The underlying software, however, still does not operate in a way that truly models real-world sytem behavior.

Object-oriented programming is a concept developed in the 1970s. With object-oriented programming, data and the procedures that act on those data are held together as an "object." The object-oriented approach to programming provides some unique capabilities that are ideally suited for computer simulation. Current efforts to use the object-oriented approach to simulation are less than optimal due to the slow processing speed of available object-oriented languages and the difficulty of programming in an unfamiliar language.

This research combines the advantages of simulation languages written in strongly-typed procedural languages with the unique capabilities of object-oriented programming. A review of relevant literature on simulation, object-oriented programming, and related topics is presented in Chapter II.

II. REVIEW OF RELEVANT LITERATURE

A. Computer Simulation

1. Preliminary concepts

Complex systems are abundant in the world of industry, government, and a host of other environments. It is often useful to analyze these systems to plan, optimize, or otherwise modify the operation of the system. To investigate these systems, historical data related to system behavior could be collected and analyzed. If past data are not available, collection of current data is an alternative. A greater problem exists if the system targeted for study is not yet in existence. In the latter case, a model of the system could be developed. The model could then be used to represent the real system and the model behavior could be studied to predict the operation of the real system under a variety of situations. Computer simulation plays an important role in this modeling of systems.

Taha [75] states that "Computer simulation should be regarded as the next best thing to observing a real system in operation." The use of a computer simulation allows system operational data to be collected over a reduced time scale without the necessary existence of the real system. These data may then be used to calculate measures of system behavior and performance.

According to Pritsker [58], simulation models can be employed at four levels:

- As explanatory devices to define a system or problem;
- As analysis vehicles to determine critical elements, components, and issues;
- As design assessors to synthesize and evaluate proposed solutions;
- As predictors to forecast and aid in planning future developments.

Computer simulation models are often built as a mathematical representation of the system in question. Queueing theory is often the basis used in the development of the model, but is not sufficient to model the behavior of a complex system. Queueing theory can be used to study isolated components of a system, but fails to adequately represent the interactions between the various elements of the system.

Simulation typically represents the system as a whole. The end result is a model capable of tracking all the individual processes and activities in the system. Data are then collected from the simulation model to analyze in appropriate fashion.

The primary benefits of computer simulation may be summarized as follows:

- Computer simulation allows complex systems to be modeled;
- Data may be collected from the simulation for later analysis;
- Time may be scaled to allow simulations of lengthy real-world operation of a system to be simulated in a relatively short period of computer simulation time;
- Simulations of nonexistent systems can be performed;
- Alternative operation of real system behavior can be quickly examined.

2. Types of simulation

The primary purpose of a computer simulation is to allow data to be gathered about the operation of a specific system as a function of time. Computer simulations are typically categorized into two distinct types:

- Discrete simulation
- Continuous simulation

In a discrete simulation model, data are gathered from the simulation model at specific points in time, typically when a change occurs in the state of the system.

Conversely, continuous simulation requires that data are collected at very small increments in time during the execution of the simulation.

As an example of the difference between the two types of simulation, consider two systems. The first system is a ticket sales outlet as illustrated in Figure 2-1. This system is categorized as a single-server queueing system. Customers arrive in single fashion and wait in line for the clerk. In this model, changes in the state of the system can only occur when a customer arrives or when the customer completes service (buys a ticket). When either of these events occur, relative measures of system performance can be collected. Typical statistics may be the current length of the queue and the waiting time in the system. At all other times during the operation of the system, the system statistics remain unchanged, only the simulation clock (discussed later) will be affected. The system must only be observed at discrete points in time, thus the name "discrete simulation."

Consider a second system comprised of the heating system for a large commercial building. The temperature must be adjusted for each area of the building depending on the current temperature in that area. A measure of the building system efficiency might be the rate of heat loss. In this case, the

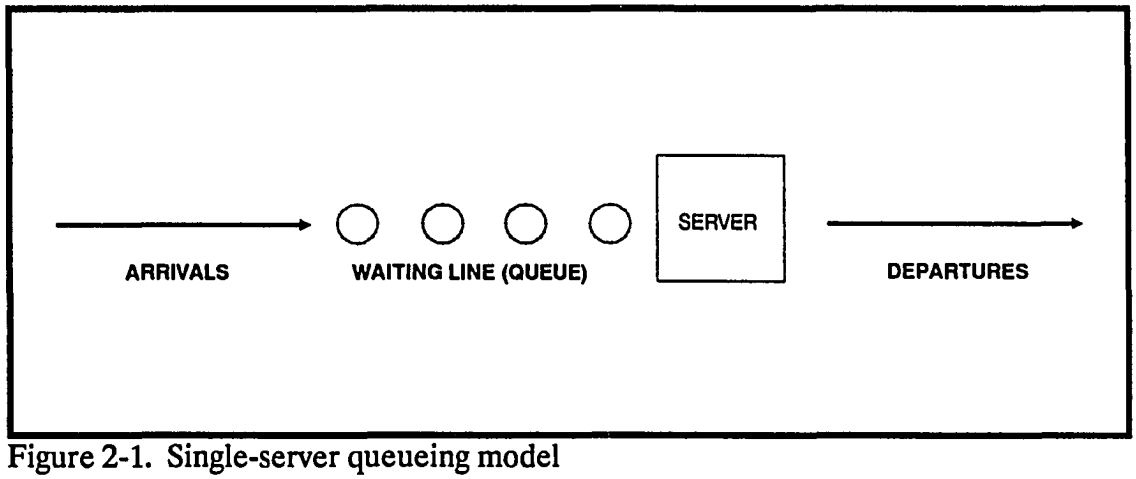

Figure 2-1. Single-server queueing model

temperature in each area must be continuously monitored. This situation would be ideal for continuous simulation. In computer simulation, it is essentially impossible to actually monitor a system continuously, so the observation of system dynamics is performed at small equal intervals of time.

Parameters of the simulation model in question sometimes dictate that both discrete and continuous modeling concepts must be utilized. The term "combined simulation" is used to describe simulation models built with both discrete and continuous simulation features.

The real world is replete with examples where both discrete and continuous simulation models are appropriate. Most continuous systems can be adequately modeled through mathematical approaches. This research emphasizes discrete simulation. The remainder of this discussion will concentrate on the particulars of discrete event simulation.

3. Discrete event simulation

As described previously, simulation models may be either discrete, continuous, or a combination of the two, depending on the manner in which change occurs in the variables of interest in the model. In most simulations, time is the independent variable. Other variables in the system are functions of time and are dependent variables. In discrete event simulation, statistics are collected from the system by monitoring the state of the system over time.

To facilitate the collection of observations, simulations must maintain a "simulated clock." Because the state of the system can only change when an event occurs, an accurate picture of the system may be obtained by advancing the simulated clock from one event time to the next. The use of the simulation clock in

this manner is called the "next event approach" and is used in most simulation languages.

According to Pritsker [58], a discrete event model can be formulated by:

- Defining the changes in state that occur at each event time;
- Describing the activities in which the entities in the system engage;
- Describing the process through which the entities in the system flow.

Three key terms used in the discussion of discrete event simulation may now be defined with reference to Figure 2-2:

- Event - An occurrence which takes place at a discrete point in time which marks the beginning or end of an activity.
- Activity - The time passage that occurs between the begin and end events.
- Process - A chronological sequence of events encompassing one or more activities.

4. World views

Simulation models are often described in terms of their "world view" in relationship to the concepts of event, activity, and process. In order, the terms used

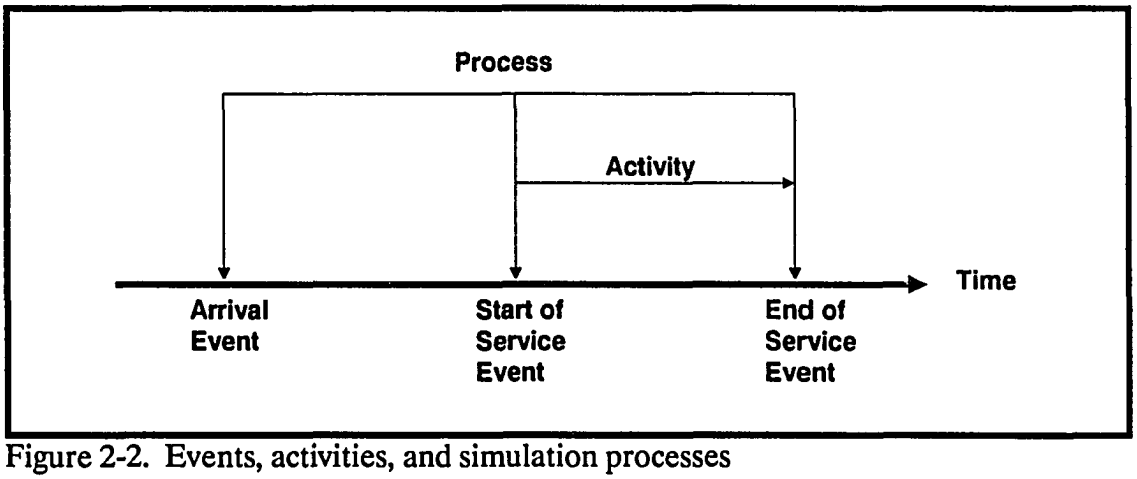

Figure 2-2. Events, activities, and simulation processes

to describe alternative world views are event, activity scanning, and process orientation.

If the world view is event-oriented, the system is modeled by defining the changes that occur at event times. The system modeler must define the events that will change the state of the system and then develop the appropriate simulation logic to correctly trigger events in a time-ordered fashion and collect the system state variables at the event times.

To illustrate the event-oriented world view, again refer to Figure 2-1. Customers arrive and enter the waiting line. When the ticket salesperson is available, the next waiting customer receives service and then exits the system. The events in this system are then:

- Arrival of a customer
- Start of service
- End of service

The state of the system remains unchanged except when one of the aforementioned events occurs. The entire system can be described in terms of these events. The simulation clock is used to trigger an event. The simulation model logic is responsible for scheduling the times that future events will occur. This schedule of events is called the "event calendar." The advantage of the event-orientation is that the dynamic behavior of the system can be observed by examination of the system variables only at the event times. Since the number of discrete events is usually limited in relationship to the total simulation time, the model is generally simpler to construct.

In a simulation system built from the activity scanning orientation, the activities are described and the conditions which cause these activities to start and end are defined in the simulation logic. In the activity orientation, the simulation logic is no longer responsible for scheduling the events on an event calendar. Instead, as the simulation clock is advanced the pre-defined start and end conditions for activities in the model are scanned. If the conditions are met, the corresponding action for the activity is initiated. When the activity scanning orientation is implemented with a standard procedural language, each activity must be scanned when the simulation clock is advanced.

Because each activity must be scanned at every clock advance, the activity orientation becomes inefficient for most simulation modeling problems. However, some aspects of the activity orientation are useful and are utilized in part by many simulation languages. In particular, many languages group standard sets of activities into single statements for inclusion in the simulation model. This approach is the process-orientation.

Process-oriented simulation languages use standardized statements to track and model the flow of entities through the system. The control logic associated with these statements is automatically executed by the simulation language. The process-oriented simulation languages are relatively simple to utilize. Processes are usually associated with symbols that describe the simulation language. The modeler need only create a network of these symbols to develop the model. Process-oriented simulation languages are ideal candidates for a graphic user interface due to their symbolic representation. However, because the simulation language is often restricted to a pre-defined set of symbols, modeling flexibility is usually less than that of the event orientation.

It has been shown that simulation languages can be grouped into categories based on the types of simulation that they perform and the view that is used in creation of the simulation model. Many simulation languages have been developed that fit each of the defined categories. The next section provides a review of some of these languages, their associated implementations, and the ongoing effort to enhance the utility of computer simulation tools.

B. Simulation Languages

1. Traditional languages

Arthur, Frendewey, Ghandforoush, and Rees [1] cite the beginning of computer simulation as the late 1950s. The original computer simulations typically consisted of FORTRAN programs written for batch operation on mainframe computers. In a recent study by Pratt [57], over 150 simulation languages were found available for microcomputers, minicomputers, and mainframe computers.

The first widely used simulation language was GPSS (General Purpose Simulation System). Developed in the 1960s, GPSS remains one of the more popular simulation languages available. Schriber [66] writes that "much of the underlying logic of discrete-event simulation is built into the GPSS simulator. Unfortunately, this language advantage becomes a disadvantage for the model builder who does not understand the simulator's internal logic, and yields to the temptation to use GPSS blindly." The same statement can be applied to most of the early simulation languages.

Another pioneer in simulation languages was SLAM (Simulation Language for Alternative Modeling). SLAM, a FORTRAN based simulation language, allows the modeler to construct simulation models based on the event, activity, or process

world views. SLAM contains facilities to support both discrete-event and continuous simulation constructs. Nearly 1000 installations of SLAM exist in academic, industrial, and governmental settings. SLAM is available for a wide variety of computers and operating systems.

The SLAM simulation languate is written in FORTRAN. Many other simulation languages and dedicated simulation programs have been developed in FORTRAN because of its widespread use and availability. Another programming language popular with developers of simulation languages is Pascal. While few complete simulation languages have been developed in Pascal, much work has been done in adding discrete-event simulation extensions to Pascal.

Frantz and Trott [24] describe the use of Pascal in the development of the Dynamic Ground Target Simulator. This system was developed to support the detailed discrete-event simulation of military activities. Pascal was used as a base language for the development of an extended language called the Model Definition Language (MDL). Features added to standard Pascal to support functions necessary for the simulation application included:

- Event-scheduling
- Message definition and output
- Scenario time
- Direct access files
- Intermodule references

Several features were excluded from the new Pascal implementation to preserve data protection and abstraction concepts. Other features were added to improve the readability of the resulting simulation code.

Smith and Smith [70] also added extensions to the Pascal language to allow management and implementation of simulations. Among the new features added to the standard language were:

- Process handling and synchronization
- List handling
- Distribution functions
- Simulation control
- Histogram functions

Hughes and Gunadi [30] added extensions to ISO standard Pascal through the development of a preprocessor that generates ISO standard Pascal as output. The additions to Pascal incorporate a mechanism for quasi-parallel scheduled processes with multiple instances. The new features of the language are for purposes of discrete-event simulations.

Barnett [3,4] describes two implementations of MICRO-PASSIM, a simulation package which provides the source code to Pascal procedures designed to allow discrete-event and continuous modeling. Among the features for simulation added to Pascal through MICRO-PASSIM are:

- Real time clock
- Queueing disciplines
- Event sequencing
- Random number generation
- Integration of continuous state variables

Seila [68] presents a similar approach to adding simulation capabilities to Pascal. SIMTOOLS is a collection of procedures and functions that allow

discrete-event simulation programs to be easily developed in Pascal. The package, which implements the event world view, has procedures for creating and deleting entities, managing lists or queues, event scheduling and sequencing, system tracing, and data collection. SIMTOOLS only provides the core for simulation in Pascal. The intent is that the user augment the routines for specialized simulation situations. The following criteria were used during the development of the package:

- Data structures and other declarations should be as simple as possible.
- Procedures and functions should be simple and descriptive and have a minimum number of parameters, generally no more than three.
- The internal mechanics of list insertion and removal, tracing/debugging output generation, and other operations should be as transparent as possible to the user, while being accessible.
- Source code should be self-documenting as much as possible.
- Standard Pascal should be used where possible.

A final reference to Pascal simulation environments is given by Thesen [77] where general information on writing simulations in Pascal is provided. An emphasis is placed on the development of efficient algorithms and data structures specific to simulation. Special attention is given to event set management and algorithms for the generation of random variates from the uniform, exponential, normal, and gamma distributions.

The previous discussion is not intended to be a complete description of traditional approaches to simulation. Other languages exist, and differ in many aspects from those mentioned. Much attention has been focused on the improvement of the user interface for simulation languages. Research and progress in the area of user interfaces are discussed in the next section.

2. Improvements in the user interface

Nance [53] indicates that simulation model representation is currently undergoing a significant transformation. The methods used for the development of simulation models had remained relatively unchanged for some 20 years. While revisions, extensions, and other conveniences have been added to the simulation languages discussed previously, no conceptual advances were obvious.

The increasing demand for simulation software spurred a concentration on improvements in the way the user interacts with the system. Generally, the improvement made in simulation software user interfaces can be grouped in four categories:

- Program generators and development environments
- Graphic input
- Graphic output and animation
- Visual interactive simulation

Kootsey and Holt [40] developed a simple user interface for the development of continuous simulation models. The user interacts with the program through a menu and is therefore insulated from the complexities of the underlying simulation model.

Favreau and Marr [21] describes the EzSIM simulation system which is designed to aid in the development of continuous simulations. The EzSIM system is primarily a database management system that contains pre-written sets of simulation commands used for continuous simulation. The user is interviewed by the system to determine the necessary components of the simulation. The required code is then generated and the simulation is performed.

Another use of the database approach is offered by Marr [49] through SIM_BY_INT. The concept of SIM_BY_INT is to interview the simulation modeler to determine the type of simulation to be performed. SIM_BY_INT would then develop a database of required information and choose from among several available simulation languages to select the most appropriate language to use. The result of this approach is that the user is not required to know how to write the actual simulation.

The interview technique is again used by Haigh and Bornhorst [27] for the NCR Corporation. The desire to simulate computer systems at NCR combined with a goal to reduce the costs of these simulations resulted in the development of several simulation environments. Each of these simulation systems uses an interactive interrogation of the user to develop a portion of the code required to eventually execute a GPSS simulation. Simulation model and report generating facilities have been developed as simulation aids.

Mathewson [50] reviews the concept of application program generator software. Application program generators serve to simplify the process of generating computer code by presenting the user with easily understood prompts and menus. Based on user responses, the program generator automatically generates the required computer code to execute the target program. When applied to simulation, a program generator would generate code to be used by a simulation package such as GPSS or SLAM. Shanehchi [69] presents the EXPRESS system which is an application program generator specifically designed for simulation. EXPRESS generates simulation code for execution by the SEE WHY simulation language.

Cobbin [12] describes SIMPLE_1 which uses the network approach to model building. The user can create and execute models totally within the SIMPLE_1 environment. This system is intended to be a complete simulation environment which supports the simulation modeling tasks of:

- Data collection
- Data analysis
- Model development
- Compilation and execution of simulations
- Analysis of simulation results

Another integrated simulation environment, TUTSIM, is described by Meerman [52]. TUTSIM is a simulation tool for the simulation of continuous dynamic systems. The model input is in dialog form, results are presented graphically, and calculations can be interrupted at any time.

The GPSS simulation language described earlier has been implemented on microcomputers. Karian and Dudewicz [36] and Cox and Cox [18] describe GPSS/PC as an interactive implementation of GPSS which operates on IBM PC compatible microcomputers. In GPSS/PC, the older multiphasic designs have been replaced by a single, integrated simulation environment that combines the functions of editing, compiling, simulating, and debugging.

Karian and Dudewicz [36] also present the PC SIMSCRIPT simulation system. Through the use of SIMLAB, a specially designed simulation laboratory environment, the user is able to interact with the simulation language processor. A prior description of SIMSCRIPT is given by Johnson, Rector, and Mullarney [33].

The acceptance of program generators and integrated simulation environments for simulation languages emphasizes the continuing need for improved user interfaces. Graphics provide another method of interaction with the user.

The use of graphic symbols for the design phase of the simulation model is implemented by Hoover [29] with MICRO-SIM, a network-based simulation system. Source and sink nodes are placed in the simulation network using graphic representations on the computer screen. Other types of nodes implemented in the system are intermediate, probabilistic, shortest queue, sequential attempt, and rotating discharge nodes.

Wadsworth [82] examines the use of graphics for both input and output in a simulation environment. MICRO-PASSIM with graphics includes both input and output graphics. Hollocks [28] further examines the relative benefits of graphics in simulation. Hollocks states that real representation of the simulation problem is maintained by the underlying simulation system. The use of graphics can substantially enhance the interface with the user. With graphics, the user can see the model and relate to the simulation. The simulation may be better understood if visualized. Graphics also allows a higher level of user interaction with the simulation.

Smith and Platt [71] reinforce the advantages of graphics in simulation, specifically in the use of animation to display the simulation in progress. Animation provides better understanding of the simulation for the model builder, the model user, and to those who wish to examine the results of the simulation.

Barta [5] describes three projects involving animated graphic output of simulation results. The intent of the projects was to determine future equipment

needs related to simulation. Grant and Weiner [26] present a discussion of factors to consider when selecting simulation systems when animation is desired.

Birtwistle, Wyvill, Levinson, and Neal [9] examined a specialized application of computer animation in simulation of distributed simulation systems. Magnenat-Thalmann and Thalmann [47] also used animation in the development of a unique computer animation language. Langlois [41] developed another computer animation language called SIMSEA which can be used to visualize a simulation.

Johnson and Poorte [34] and Magnenat-Thalmann and Thalmann [46] propose some standards to follow in the development and implementation of animation in simulation software.

The use of graphics for simulation model input and animated output was first examined in detail by Hurrion [31]. Hurrion coined the term "Visual Interactive Simulation (VIS)" to describe the concept of a simulation system that would utilize graphics for both input and output. Macintosh, Hawkins, and Shepherd [44] further describe the development of a VIS philosophy at Ford of Europe. Bell and O'Keefe [6] review the use of VIS in the United Kingdom and North America.

While not called Visual Interactive Simulation systems, The Extended Simulation System (TESS) and GPSS/PC can be appropriately described as VIS implementations. Standridge [73] and Cox [17] describe each of these simulation languages. TESS provides an integrated environment for performing simulation projects in SLAM and includes the capabilities to graphically build SLAM networks, enter and manage simulation data, prepare reports and graphs, analyze simulation results, and animate simulation runs. The latest version of GPSS/PC utilizes interactive graphics and animation in its simulation environment.

Clearly, the user interface component of simulation languages has undergone a great deal of change since the early simulation languages first became available. Many other advances in simulation technology have occurred during the same period that are not as obvious. An important area of current research involves the development and implementation of object-oriented simulation languages. The concept of object-oriented programming is discussed in the next section.

C. The Object-Oriented Programming Paradigm

1. Historical perspectives

The simulation languages reviewed previously are built on procedural languages such as FORTRAN or Pascal. The discussion now turns to a concept called "object-oriented programming."

According to MacLennan [45], Alan Kay is considered to be the principal person responsible for the development of an object-oriented programming language called "Smalltalk." In the late 1960s, Kay realized that advances in computer design technology would eventually reduce the size and price of computers to the point that it would be possible for everyone to own a personal computer of considerable power. However, existing computer languages were designed for the mainframe computer experts. Kay thought that the absence of an adequate programming vehicle for these small computers may be an impediment to the success of personal computers.

Kay investigated simulation and graphics-oriented languages as a new programming medium. He then proposed the concept of a small computer called "Dynabook" to Xerox Corporation. In 1971 the Xerox Palo Alto Research Center began a research project to develop the Dynabook. Smalltalk-72, the language for

the Dynabook, was designed and implemented by 1972. The Smalltalk language has been revised several times and is still undergoing change.

Smalltalk remains as one of the most popular implementations of the object-oriented programming philosophy. The Smalltalk programming language is entirely object-oriented. Actor, described by Duff [20], is another example of a programming language that is exclusively object-oriented. Stein [74] presents the OPAL object-oriented programming language.

Other languages have been extended to include object-oriented tools. According to Cornish [14], the C++ preprocessor, the Flavors system for LISP machines, and the Common LISP Object System are examples of languages that not only provide standard programming features, but also include object-oriented programming features. Pountain [56] describes object-oriented extensions that have been added to the FORTH programming language.

Many implementations of object-oriented programming languages are available to build object-oriented applications. The next section provides a discussion of the nature of object-oriented programming. Because the term "object-oriented programming" was first used to describe the Smalltalk language, the following discussion will present the concepts of object-oriented programming from the Smalltalk perspective.

2. Elements of object-oriented programming

Cornish [14] states that object-oriented programming is not another programming language. It is a set of programming techniques that can be used in many programming languages. The term "object-oriented programming" has been incorrectly used to describe many of the graphic user interfaces found in modern microcomputer applications such as GEM and Microsoft Windows. While it is true

that many of the implementations of object-oriented programming are based on a graphic environment, graphics are not part of the object-oriented philosophy.

Most computer languages operate under the "data-procedure" paradigm. Procedures (distinct sections of computer code) act on data passed to them. Procedures must be prepared for every type of task required by the resultant program. An example would be comparison function, compare(X1,X2), that takes two parameters and returns a value indicating whether the first parameter is less than, equal to, or greater than the second parameter. In a strongly-typed language such as Pascal, separate compare functions would be prepared for each data type that requires comparison.

Object-oriented languages employ a data or "object-oriented" approach to programming. Instead of passing data to procedures, the data (objects) are asked to perform operations on themselves through the use of "messages." Using the comparison function example, an object-oriented program statement might appear as follows:

X1 : compare X2.

In this example, the object X1 is asked to perform the compare function on itself. In this case, X1 is said to be the "receiver" of the message "compare" and X2 is supplied as an argument object.

Figure 2-3 illustrates the basic terminology used in object-oriented programming for the compare function example. X1 and X2 are "instances" of a "class." The class provides all the information necessary to construct and use objects of a particular kind, its instances. Each instance belongs to one class, but a class may have multiple instances.

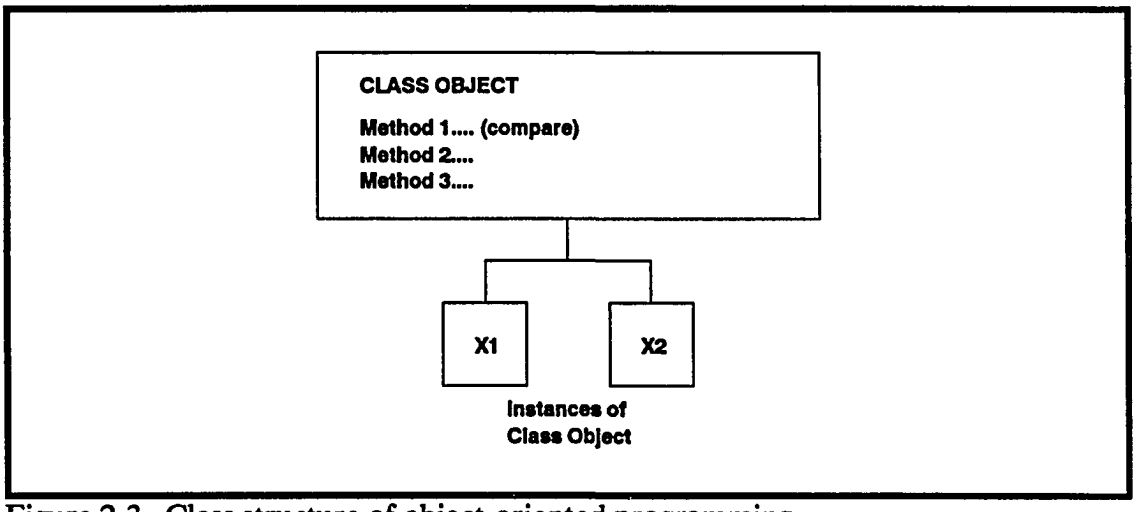

Figure 2-3. Class structure of object-oriented programming

The class also provides storage for "methods." Methods are simply procedures designed to operate on instances of a class. In the example, compare would be a class method. Methods are invoked by sending "messages" to an instance of a class. Each instance of a class has storage allocated to it to maintain its individual state. The state of an object is referenced by its "instance variables."

Computation in an object-oriented system is achieved by sending a message to an object which invokes a method in the object's class. In the example, the message "compare" is sent to the object "X1", which invokes the compare method in the class object. Typically, a method will send messages to other objects. Each message-send eventually returns a result to the sender. The state of some of the objects in a message-send chain may change as a result of the activity. Much of the message sending that occurs in an object-oriented system is automatic and transparent to the user.

According to Pascoe [55], a programming language must have four elements to support the object-oriented programming philosophy:

- Information hiding
- Data abstraction
- Dynamic binding
- Inheritance

Tesler [76] describes each of these terms in detail. Information hiding refers to the breaking up of programs into modules that can be modified independently. In an object-oriented system, every module is an object, that is, a data structure that contains the procedures that operate on it. In designing an object-oriented program, objects are identified which constitute a useful portion of the problem at hand. The objects contain their own data, and hide that data from other objects.

Data abstraction is the process of hiding data structures within objects. This practice avoids the strong type-checking requirements of many programming languages. Data structures may be dynamically modified without requiring changes to the underlying computer code. Procedures within the object act on the data independent of the type. These procedures are called "methods" in the object-oriented programming paradigm. Dynamic binding occurs when the object-oriented program is executed. Only messages are sent to objects and the data types and methods are determined by the object. This is known as "polymorphism."

Object-oriented languages share code through "inheritance." A new object may be created as a variation or exact copy of an existing object. The new object is called a subclass of the old class, and the old object is a superclass of the new object. Objects in the subclass inherit all the properties of the superclass, including the

implementations of methods. The subclass can define additional methods and redefine old methods.

3. Advantages of object-oriented programming

Object-oriented programming offers many advantages over procedural languages. Information hiding and data abstraction increase reliability and help separate the specification of procedures and data types from implementation. Dynamic binding increases the flexibility of the program by permitting the addition of new classes of objects (data types) without having to modify existing code. The addition of inheritance to dynamic binding permits code to be reused with minimal effort. In general, this will reduce the size of the program code and increase programmer productivity. Object-oriented programs are typically easier to maintain because of the direct relationship between data and procedures.

Another important advantage of object-oriented languages is the correspondence between objects in the language and real-world entities. The programmer may find fewer obstacles in the design phase of a programming project when the program design closely approximates its real-world counterpart.

4. Disadvantages of object-oriented programming

Object-oriented languages have some characteristics that are considered to be disadvantages by some. The dynamic binding mechanism of late-binding object-oriented languages usually requires a high level of computer processor overhead. A message-send takes more time than a standard function call. The comparison between message sends and function calls is difficult to measure. While the message-send is slower, it usually accomplishes more than a function call.

Another disadvantage is that the implementation of the object-oriented language is often more complex than a comparable procedural language. The

programmer must often learn an extensive class library before becoming proficient in an object-oriented language.

In the final analysis, the choice of programming environments is related to a multitude of factors, only some of which were considered here. One area that appears to be well suited for the application of object-oriented languages is simulation. The use of object-oriented languages in computer simulation is discussed in the next section.

D. Object-Oriented Simulation

1. Knowledge-based simulation and the DEVS formalism

In recent years an important concept in simulation research known as "knowledge-based simulation" has been developed and discussed by Zeigler [83, 84, 85], Zeigler and Tag Gon [86], Rozenblit and Zeigler [61], Rozenblit, Suleyman, and Zeigler [62], Ruiz-Mier, Talavage, and Ben-Arieh [63], and Concepcion [13]. These researchers noted that many concepts related to simulation were also present in the design and implementation of artificial intelligence systems. This similarity provoked a realization that the two sciences of artificial intelligence and simulation may someday merge. In preparation for the possible merger, the researchers decided that the concepts related to both sciences should be studied in an effort to define a cohesive approach to knowledge-based model preparation and design. In particular, Zeigler advanced the concept of the Discrete Event Simulation Specification (DEVS formalism) as a standard approach to knowledge-based simulation system design and implementation.

Figure 2-4 illustrates the fundamental concepts of modularity and model bases. Suppose that model A and model B are in the model base. If the models are

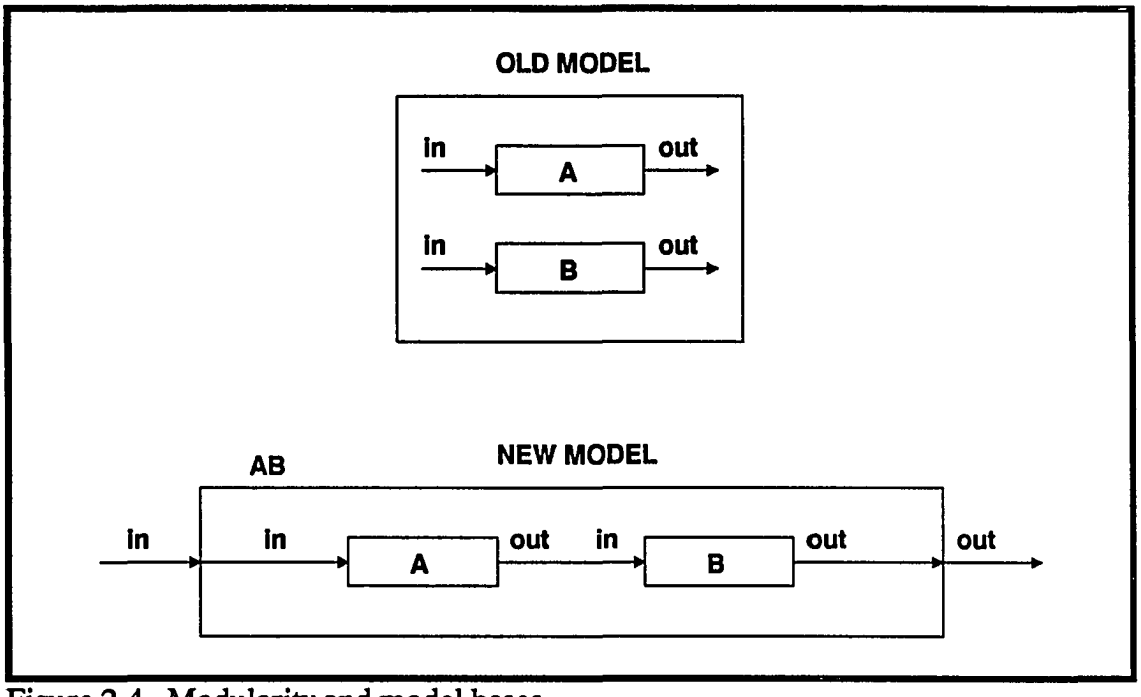

Figure 2-4. Modularity and model bases

in proper modular format, it would be possible to create a new model by specifying the form of inputs and outputs of A and B that are to be connected to each other and to external ports, an operation called "coupling." The resulting model, called AB, would again be in modular form. The coupling process could then continue to build an unlimited variety of models. The model components would be modular and hierarchical.

An important benefit of the modular, hierarchical approach is that a model in the model base can be independently tested by coupling a test module to it. The result is a reliable and efficient verification of large simulation models.

An important concept in the DEVS formalism is the "coupling specification." There are three parts in the coupling specification:

- External input coupling - describes how the input ports of the composite model are identified with the input ports of the components.
- External output coupling - tells how the output ports of the composite model are connected to the output ports of the components.
- Internal coupling - specifies how the components inside the model are interconnected.

In general, the coupling relationships of the model components are illustrated with a composition tree. By following the limbs of a composition tree, a submodel composition may be obtained. This submodel decomposition supports the modular, hierarchical concept.

The specification of a modular discrete-event simulation model requires a different view than that taken by traditional simulation languages. As described previously, a model must be viewed as possessing input and output ports through which all interaction with the environment flows. In the case of a discrete-event model, events determine values present on the input and output ports of the model components.

A pseudo-code has been developed to assist in the specification of discrete-event models. This code uses the form "when receive x on input port p, send y to output port p." This is known as a transition statement and is similar to the form of predicate logic used in many expert system languages. In addition, the modular concepts of the DEVS formalism relate closely to the data abstraction and modularity concepts present in object-oriented languages. Additional control statements added to the DEVS pseudo-code permit complete specification of the desired model.

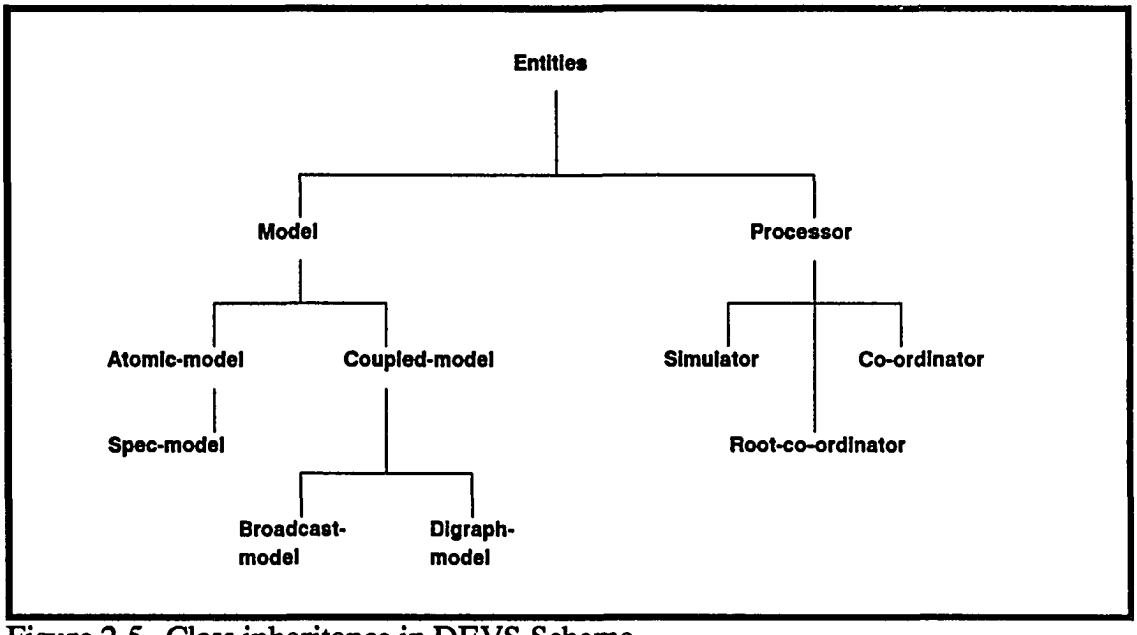

Figure 2-5. Class inheritance in DEVS-Scheme

Zeigler [84] describes the implementation of the DEVS formalism for discrete-event modeling in PC-Scheme, an object-oriented LISP dialect for microcomputers. In contrast to existing knowledge-based simulation systems, DEVS-Scheme is based on the DEVS formalism discussed previously. DEVS-Scheme is a shell that operates in conjunction with PC-Scheme in such a way that all the underlying object-oriented and LISP features are available to the user.

DEVS-Scheme is primarily coded in SCOOPS, the object-oriented superset of PC-Scheme. Figure 2-5 illustrates the class inheritance structure of DEVS-Scheme. The entity object provides all the tools for manipulation of objects. The model and processor classes provide the basic constructs required for modeling and simulation.

The atomic-model class implements DEVS formalism for discrete-event models. Spec-model class objects contain the specific entity definitions and port specifications of the hierarchical model. Coupled-models is the major class which embodies the composition constructs of the DEVS formalism. Digraph-models and broadcast models are specializations of the coupled-model class which enable specification of coupled-models in special ways with linked and finite set components.

The simulators and co-ordinators are special classes of processors which carry out the simulation of DEVS models by implementing the abstract simulator principles developed as part of the theory. Simulators and co-ordinators are assigned to handle atomic-models and coupled-models in a one-to-one manner, respectively.

Simulation in a DEVS model proceeds by means of messages passed among the processors which carry information concerning internal and external events, as well as data required for synchronization. DEVS-Scheme runs interactively; a simulation run can be interrupted during the root-coordinator's cycle so that a pause occurs only at a valid model state. The simulation can be restarted from the resulting state after desired modifications have been made to the model.

Aside from a minimal standardization of the interfaces, DEVS-Scheme does not impose any particular choice of typing of the input, state, and output objects. Because DEVS-Scheme is based on LISP, objects are represented as lists constructed and decomposed using cons, car, and cdr functions. For these functions, there are no types, therefore no type specification is required of DEVS-Scheme. This facet offers generality, but allows no strong type-checking, which becomes the responsibility of the programmer. Thus, large memory requirements, the slow

execution of languages such as LISP, and the requirement of user-facilitated type-checking negate many of the benefits of the DEVS-Scheme language for discrete-event modeling and simulation.

While many researchers approached simulation from the knowledge-representation viewpoint, others made attempts to advance traditional views of simulation. A promising area of research related to object-oriented simulation is the topic of process-oriented simulation discussed in the next section.

2. Process-oriented simulation

An often misused and misunderstood term in computer simulation is "process-oriented." This term was used previously to define a "world-view" taken by some simulation languages. While correct, that usage does not represent the complete definition of process-orientation. A more general description of process-oriented simulation languages and their implementation is reviewed in this section.

Golden [25] elaborates the software engineering principles required in a process-oriented simulation language. Many of the concepts presented are utilized in the current research involving process-orientation. In a process-oriented simulation language, modeled systems are viewed as a collection of interacting processes. Some of the processes are separated into subprocesses to facilitate simplicity, modularity, and ease of programming. Saydam defines the properties of a process as:

- It behaves as a separate, independently controlled program.
- It has a well-defined behavior algorithm.
- It is capable of generating objects (entities) and processing them or passing them into other processes.

- It can be activated, put on hold, or terminated at desired points in time or based on certain conditions.
- Once activated, a process repeats its behavior until it is put on hold or terminated.
- Many copies (instances) of the same process can be obtained and may be initiated to work in parallel.

The reader may note that the previous definition of process-orientation does not directly match that of the process world view defined earlier, but a closer inspection reveals some similarity. The process world view does indeed represent a simulation model as a group of activities (processes) and operates on them as a related group. Traditional simulation languages attempted to implement the process world view by using activity scanning in an effort to achieve parallelism in operation of the simulation model.

Original simulation languages as described by Banks and Carson [2] could not generate new processes and could not, in reality, achieve the parallelism sought in process-oriented simulation, but the use of high-speed computers could approximate that behavior. Recent research offers other approaches to process-oriented simulation, but the underlying concepts remain unaltered. The primary change of direction has been in the development and extension of different computer languages to advance the process-oriented approach.

Decker and Maierhofer [19] describe a simulation language called BORIS which represents an attempt at process-orientation with a strongly-typed procedural language (Pascal). The building-block approach used in BORIS, along with the separately compiled modules available in Pascal does provide some of the constructs of a process-oriented simulation language. However, the approach used with

BORIS forces the use of strong types in the definition of objects (processes) and cannot dynamically create these objects. Parallelism is not achieved in BORIS and the concept of separate operational modules is not well supported.

Hughes and Gunadi [30] used Pascal as a base language to implement parts of process-oriented simulation. Extensions were added to standard Pascal for discrete-event simulation with mechanisms for quasi-parallel execution of scheduled processes with multiple instances. The major drawback of this implementation is the strict use of a preprocessor to generate native code for the following compilation step. The objects and their multiple instances are created at compile time and cannot be interrupted or modified during the execution of the simulation.

In addition, the programmer must rely on strong types to define the processes and must generate the appropriate event scheduling code prior to compilation. In general, the preprocessor, not the resultant simulation program, handles the process-oriented aspect of the simulation. Malloy and Soffa [48] use Pascal as the base language for SIMCAL, a merger of Simula and Pascal. SIMCAL uses the preprocessor approach and does little to add to the flexibility and ease of programming desired of process-oriented simulation languages.

Another attempt at process-oriented simulation in Pascal is offered by Vaucher [81] with the PSIM simulation language. Procedures were developed to facilitate object creation, scheduling, and interaction. Again however, the programmer is primarily responsible for the proper creation and scheduling of processes, the resulting simulation model cannot be altered during execution (after compilation), and many of the key concepts of a process-oriented language are not implemented.

A similar approach is taken by L'Ecuyer and Giroux [42] using Modula-2, a language similar to Pascal. The SIMOD language utilizes a structured set of precompiled modules for scheduling and process interaction. Modula-2 implementation of process-orientation does little to alleviate programmer involvement in the preparation of a simulation model and only partially supports a complete set of process-oriented simulation facilities.

The C language has been used by Schwetman [67] to implement a partially process-oriented simulation package. In addition to supporting process-oriented simulation, CSIM supports features dealing with modeling system resources, message passing, data collection, and debugging. Like many previous attempts, CSIM offers many process-oriented features but requires much of the programmer.

Current research in process-oriented simulation is turning toward symbolic programming languages as an alternative to traditional languages. Stairmand and Kreutzer [72] describe the use of LISP to develop a process-oriented simulation system called POSE. The use of LISP as a base language offers the desired interactive flexibility and list processing capabilities.

While previous research has included parts of the object-oriented paradigm, the concentration is now on the full implementation of the object-oriented approach to simulation as reviewed in the next section.

3. Implementations of object-oriented simulation

The use of the object-oriented paradigm in simulation is documented by McFall and Klahr [51] in their discussion of Rand Corporation's ROSS language. ROSS is an object-oriented simulation language used primarily in the area of military war-game simulation. This language was one of the first to provide

inheritance from multiple classes of objects, a feature that is well proven in other areas of knowledge-based programming.

Smalltalk is an object-oriented programming language based on Simula, an extension of Algol intended for simulation. Smalltalk objects are well suited to modeling real-world objects. Specifically, the data values inside an object can represent the properties and relations in which that object participates, and the behavior of the Smalltalk object can model the behavior of the real-world object. Therefore, in Smalltalk, the dominant paradigm of programming is modeling or simulation. Because of its close relationship with simulation, Smalltalk has been used in many simulation applications.

Knapp [38, 39] describes one of many possible Smalltalk simulation environments. Everything is Smalltalk is an object which is an instance of a class. Each class contains class variables, templates for instance variables and the instances themselves, and methods (procedures) for processing messages sent to objects of that class. In Smalltalk execution proceeds through objects sending messages to other objects and waiting until the other objects reply. The application of these concepts to simulation is apparent.

Users of Smalltalk have extended the original language to provide classes for discrete-event simulation. The user may utilize these classes directly or extend them through the subclass mechanism to control the simulation. The simulation classes include Simulation, SimulationObject, DelayedEvent, WaitingSimulationObject, Resource, ResourceProvider, and ResourceCoordinator. There are also classes to provide the necessary probability distributions.

Ulgen and Thomasma [78] further describe the Smalltalk simulation environment and compare simulation in Smalltalk versus traditional languages. Eight features are compared:

- Modeling orientation
- Input flexibility
- Structural modularity
- Modeling conciseness
- Macro capability and hierarchical modeling
- Standard statistics generation and data analysis
- Animation
- Interactive model debugging

The Smalltalk simulation environment supports an object-oriented approach where for each object a set of tasks are defined. Objects perform their tasks independently and pass messages to each other to coordinate their work. This concept fits the real-world view of systems in which message passing occurs. Traditional simulation languages generally cannot support this messages passing capability. The burden of selecting the model orientation is placed on the user. Input flexibility is provided in Smalltalk simulations through the use of windows and pop-up screens for data input. Most traditional simulation languages also support some type of input aids.

Structural modularity refers to the modular organization of the simulation software. The Smalltalk environment naturally supports modularity while other simulation languages must be specially structured to support this feature. Concise simulation models are typically easier to build and debug. Many of the traditional

simulation languages, as well as Smalltalk, support conciseness through the use of block components and simulation network construction.

Traditional simulation languages do not typically support the hierarchical modeling concepts described previously. The hierarchical nature of object-oriented languages such as Smalltalk naturally implement the hierarchical approach to simulation modeling. In addition, macros of system components can easily be constructed and stored as object in an object-oriented system and are also available in many traditional languages.

An object-oriented simulation language provides no special advantage in the generation and analysis of statistics, although the graphics basis of most object oriented languages such as Smalltalk may provide a richer set of output types. Animation may also be easier to implement in a language that is already based on graphics, but animation is readily available in many traditional simulation languages. Cammarata, Gates, and Rothenberg [10] state that animation may even be more difficult in an object-oriented language. The interruptible facet of Smalltalk adds flexibility to model debugging, which is often difficult in traditional simulation languages.

Concurrency in simulation models can be readily obtained through the use of an object-oriented paradigm. Bezevin [7] discusses concurrency in Smalltalk. A simulation platform called SimTalk was built within the Smalltalk environment. Several aspects of producing simulation software were investigated including graphical programming, interactive programming, automatic tracing and statistics gathering mechanisms, and advanced programming techniques useful for simulation. King and Fisher [37] describe the development of extensions to the Smalltalk language for use in shop-floor design, simulation, and evaluation.

The concept of object-oriented simulation is not restricted to a typical object-oriented programming language such as Smalltalk. Unger [79] discusses the use of C, Ada, and Simula for object-oriented simulation with results that tend toward complexity. Samuels and Spiegel [64] report better success with Ada, but inspection of the research reveals that the end result does not incorporate many of the features required of the object-oriented paradigm and is more directed at the interactive debugging aspect of the simulation.

The research related to object-oriented simulation has primarily focused on the use of object-oriented languages such as Smalltalk. Other languages such as Pascal, Ada, and C have also received some attention. A compromise has emerged between the speed and structure of traditional languages versus the inheritance and class structure mechanisms of object-oriented languages. Parallel processes and interactive debugging facilities are desirable components of an object-oriented simulation system, yet slow execution speed and large memory requirements inhibit the large-scale use of object-oriented simulation. Clearly, much work is needed in this area. The next section presents a discussion of the future directions in object-oriented simulation.

4. Future directions in object-oriented simulation

Birtwistle [8], Jefferson [32], and Vaucher [80] each discuss their views of the future of simulation software. Rothenberg [60] specifically addresses the need for further research in object-oriented simulation software regarding modeling power, control representation, comprehensibility, and reusability of model building.

Current discrete-event simulation systems are limited in the types of questions they can answer. The simulationist typically specifies the model inputs and then runs the simulation. This corresponds to a "what if" question. Typical users

would also like to ask "why," "how," and optimization questions. The limitation of current simulation languages to answer such questions results primarily from their underlying representation of knowledge and their lack of inferential capabilities. One possible approach to add inference capabilities to simulation software is to use the "inference engine" approach applied to many expert system packages.

The control of the simulation model becomes more important as the power of the software increases. One of the major shortcomings of most simulation systems is their inability to represent models with varying degrees of aggregation. The modeler must predetermine the level of aggregation of the model, and program the simulation system accordingly. It is currently difficult, if not impossible, to vary the level of aggregation after the simulation has been started. Dynamic aggregation would allow a simulation to be run at one aggregate level to a certain point and then continued at a different aggregate level.

A related limitation is the display of the aggregate levels in usable form. Many object-oriented systems are graphics-based. Graphics-based systems are programs that rely on graphic symbols on the computer screen for user interaction. Smalltalk, Microsoft Windows, and the Apple Macintosh operating system are examples of graphics-based systems. Graphics-based systems allow greater flexibility in the presentation of objects to the user and can help the user visualize the inner workings of a simulation, but with these systems the user cannot control the level of visual interaction that occurs. The combination of dynamic aggregation and visual detail changes would be desirable. In such a system, the user could "zoom" to different detail levels of the simulation with the desired level of detail always available. In addition, the user should have full control of starting and

stopping the model at any point, whether to merely examine the current state of the system or to change the simulation parameters.

The object-oriented paradigm focuses on the definitions of objects with a built-in inheritance mechanism in the class and subclass concept. This organization is limited to the pure hierarchical model relation while real-world models are often based on many other types of relations. The object-oriented paradigm should be extended to cover other types of relations.

Object-oriented simulation systems, as well as the traditional simulation languages, often must introduce components to the simulation model that are created for the direct support of the simulation system but do not relate to a real-world object. It is desirable to reduce the quantity of artificial objects required in simulation systems in an effort to reduce complexity and distraction.

A further problem in current object-oriented simulation systems involves the scope of the simulation. Although objects are theoretically intended to encapsulate data and operations, most current environments make object names, message forms, and even attribute names globally available. Larger and more complex simulations will require that strict data hiding and abstraction principles are followed.

A final goal in object-oriented simulation research is to develop a workable object definition paradigm that would be usable for all types of objects. Such a representation could be implemented in database form and accessed by the model builder to create complex models from a standard, albeit large database of objects.

5. Summary

The preceding review of literature represents a diverse array of information related to simulation. A brief history of computer simulation was presented, followed by a review of traditional simulation languages and improvements that

have been made in the user interfaces of these languages. The concept of object-oriented programming preceded a discussion of object-oriented simulation and related topics.

This research furthers the development of the object-oriented paradigm in simulation. The next chapter presents a detailed discussion of the object-oriented simulation software developed in this research.

III. SIMULATION SOFTWARE DESIGN

A. Introduction

This chapter provides a discussion of the design of the object-oriented simulation software. As stated previously, the primary goal of this research is to develop object-oriented extensions for simulation in a strongly-typed procedural language. There are several distinct goals to be achieved during the course of the research.

A programming language must be selected to serve as the basis for the research. Next, the fundamental programming algorithms and procedures must be developed. Class and instance creation and manipulation must be incorporated into the software. The object-oriented extensions for simulation must then be added to the base program. The remainder of this chapter presents a discussion of each of these major research activities.

B. Language Selection

Many alternative languages are potential candidates fo this research. The major requirement is that the language used must be a strongly-typed procedural language. Traditional simulation languages are typically built in FORTRAN, while modern approaches often utilize languages such as Ada, C, and Pascal.

The use of a strongly-typed language avoids the late binding of data types inherent in languages without strong typing. Binding is the process of allocating memory locations for program data. The size and structure of these memory locations depends on the data types. If the data types are known at compile time, early binding may be performed, thus reducing the execution time of the program.

The use of a procedural language allows increased program modularity which reduces code maintenance. Modularity also allows greater use of common code throughout the software. In addition, new operating systems are constructed with procedural languages. If the software created in this research is to eventually be ported to new operating systems, the use of a procedural language will reduce future portability problems.

The object-oriented nature of this research indicates that a programming language that already contains some type of object-oriented extensions would be useful. Languages such as Objective C or C++ would meet this criterion. However, in an effort to construct the object-oriented portion of the simulation language from an unbiased viewpoint, the programming language used should be one without object-oriented extensions.

Strong type checking exists in many of the modern programming languages such as C, Pascal, and Ada. The desired procedure orientation is also present in these languages. The remaining criteria for programming language selection are:

- Fast compilation to minimize program development time.
- Integrated environment to provide ease of use and maximum programmer productivity.
- Capability to utilize external assembly language subroutines to allow for the advanced programming requirements necessary in this research.
- Interrupt support to enable the use of multitasking primitives for error handling.
- Availability on MS-DOS microcomputers to fit the equipment available for this research.
- Commonly used language to extend the comprehensibility of those who later examine or extend this research.
- Separate module compilation to allow a unitized approach to the construction of the simulation language.

After consideration of the listed criteria, Borland International's Turbo Pascal version 5.0 was selected as the language to use for this research. Turbo Pascal provides an integrated environment with built-in debugging facilities. Compilation with Turbo Pascal is fast and the language supports external assembly language subroutines. Full interrupt support is available and the language operates on MS-DOS microcomputers. Separate module compilation allows data hiding necessary with the object-oriented paradigm.

After selection of the programming language, the next step is to develop the overall structure of the simulation language with respect to the object-oriented paradigm. The next section provides an overview of the simulation program structure.

C. Simulation Program Structure

The software created in this research is capable of simulating systems with multiple servers and queues. Arrival and service time distributions may be selected from the uniform, exponential, and normal family of distributions. Resource usage is not supported in the simulation program. Figure 3-1 shows the general structure of the simulation software developed in this research. As indicated, the program will consist of three major sections.

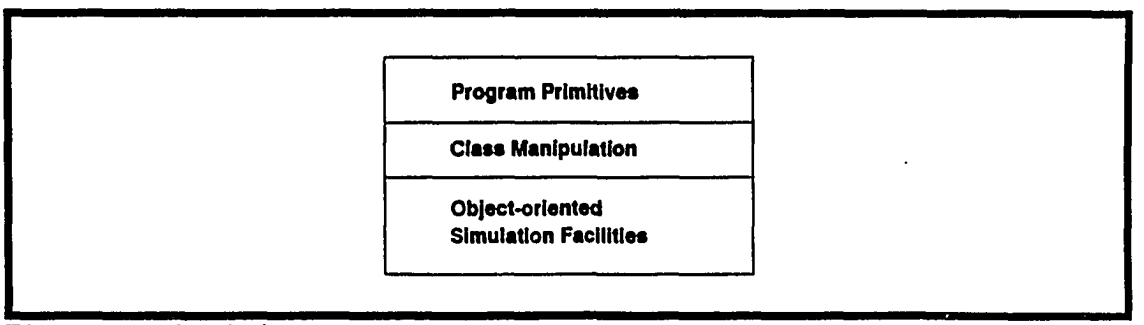

Figure 3-1. Simulation program structure

Many low-level routines are required in the program that are not directly related to simulation. These low-level routines are referred to as "program primitives." The concept of "classes" in object-oriented programming is key to the proper development of the program. Class manipulation forms the second major component of the software. The third major component of the simulation software developed in this research is devoted to object-oriented simulation facilities which handle the simulation proper.

Each of the three major components of the software are discussed in detail in the following sections. The next section reviews the program primitives.

D. Program Primitives

The development of any computer program requires the preparation of many facilities of a general nature. The program for this research also requires numerous program primitives for the successful implementation of the complete program. The program primitives can be placed in several categories which include:

- Keyboard handling
- Screen input/output
- Printer output
- Error handling
- Miscellaneous routines

It has been said that 90% of most computer programs are dedicated to the handling of input and output. The program for this research also makes extensive use of input and output through the computer display and the keyboard. In an effort to achieve maximum program speed and efficiency, routines for input/output received much attention during program development.

```
function Getakey:byte;
{ get a keystroke from the user }
var
    Regs : registers;
begin
    repeat   { wait for a keystroke }
    until KeyPressed;
    Regs.Ax:=$0000;     { read the keyboard, something has been pressed }
    Intr($16,Regs);
    if Lo(Regs.Ax)=$00 then
        Getakey:=128+Hi(Regs.Ax)       { add 128 if special key }
    else
        Getakey:=Lo(Regs.Ax);
end;


function KeyBoard(OkSet:MenuSet; Cursor:byte):byte;
{ gets a valid keystroke and optionally runs pop-ups }
var
    Ch:          byte;
    OldCursor:   byte;
    Regs:        registers;
begin
    OldCursor:=CurrentCursor;
    SetCursor(Cursor);
    repeat
        Ch:=Getakey;
        if not (Ch in OkSet) then Beep;     { beep if invalid }
    until (Ch in OkSet);
    KeyBoard:=Ch;            { valid key was selected - return keystroke }
    SetCursor(OldCursor);
end;
```

Figure 3-2. Keyboard handling routines

1. Keyboard handling

The user of the program will be required to enter data from the keyboard. Routines to handle keyboard input are an essential part of the program. The Pascal language provides basic keyboard input through the READ function. The READ function does not allow input of function keys and does not allow for strict error checking and confinement of user input to a restricted set of allowable characters. The GETAKEY and KEYBOARD functions shown in Figure 3-2 replace the

```
FWAttr      EQU   BYTE PTR [BP+6]
FWCol       EQU   WORD PTR [BP+8]
FWRow       EQU   WORD PTR [BP+10]
FWSt        EQU   DWORD PTR [BP+12]

FastWrite   PROC  FAR

            PUSH  BP              ;Save BP
            MOV   BP,SP           ;Set up stack frame
            PUSH  DS              ;Save DS
            MOV   AX,FWRow        ;AX = Row
            MOV   DI,FWCol        ;DI = Column
            CALL  CalcOffset      ;Call routine to calculate offset
            MOV   CL,RetraceMode  ;Grab this before changing DS
            LDS   SI,FWSt         ;DS:SI points to St[0]
            CLD                   ;Set direction to forward
            XOR   AX,AX           ;AX = 0
            LODSB                 ;AX = Length(St); DS:SI - St[1]
            XCHG  AX,CX           ;CX = Length; AL = Wait
            JCXZ  FWExit          ;If string empty, exit
            MOV   AH,FWAttr       ;AH = Attribute
            RCR   AL,1            ;If RetraceMode is False...
            JNC   FWMono          ; use "FWMono" routine
            MOV   DX,03DAh        ;Point DX to CGA status port
FWGetNext:
            LODSB                 ;Load next character into AL
            MOV   BX,AX           ;Store video word in BX
            CLI                   ;No interrupts now
FWWaitNoH:
            IN    AL,DX           ;Get 6845 status
            TEST  AL,8            ;Vertical retrace in progress?
            JNZ   FWStore         ;If so, go
            RCR   AL,1            ;Else, wait for end of
            JC    FWWaitNoH       ; horizontal retrace
FWWaitH:
            IN    AL,DX           ;Get 6845 status again
            RCR   AL,1            ;Wait for horizontal
            JNC   FWWaitH         ; retrace
FWStore:
            MOV   AX,BX           ;Move word back to AX...
            STOSW                 ; and then to screen
            STI                   ;Allow interrupts!
            LOOP  FWGetNext       ;Get next character
            JMP   FWExit          ;Done
FWMono:
            LODSB                 ;Load next character into AL
            STOSW                 ;Move video word into place
            LOOP  FWMono          ;Get next character
FWExit:
            POP   DS              ;Restore DS
            MOV   SP,BP           ;Restore SP
            POP   BP              ;Restore BP
            RET   10              ;Remove parameters and return

FastWrite   ENDP
```

Figure 3-3. Assembly language routine for screen output

```
procedure WriteFast(X,Y,SC:byte; S:string);
{ use fastest possible write routine to write a string at X,Y in SC color }
begin
    FastWrite(S,Y,X,SC);
end;


procedure WriteAt(X,Y:byte; S:string);
{ write a string at X,Y with specified imbedded colors (default is norm) }
var
    Attrs:  array [0..6] of byte Absolute BackC;
    CAttr:  byte;          { current attribute }
    Ps:     byte;          { current position }
    Len:    byte;          { length of string }
begin
    if Pos(#255,S) = 0 then begin
        FastWrite(S,Y,X,NormC);
        Exit;
    end;
    CAttr: = NormC;      { default to normal text }
    Ps: = 0;
    Len: = Ord(S[0]);
    while Ps > 0 do begin
        Inc(Ps);
        if S[Ps] = #255 then begin      { special color attribute }
            CAttr: = Attrs[Ord(S[Succ(Ps)])];
            Inc(Ps,2);
        end;
        FastWrite(S[Ps],Y,X,CAttr);
        Inc(X);
    end;
end;
```

Figure 3-4. Pascal interfaces to screen routine

READ function by reading keyboard input directly from the host machine's low-level keyboard buffer to provide the desired functionality.

2. Screen output

A large amount of information is manipulated during a simulation program. One of the major thrusts of this research is to develop software that is highly visual in an effort to demonstrate the object-oriented functions of the program. Pascal provides screen output with the WRITE procedure but this facility is inefficient and too slow for this research. The routines shown in Figure 3-3 and Figure 3-4 were

```
function WritePrt(S:string):boolean;
{ print and check for errors or user abort }
const
    PWait   =   20000;  { 20 second wait for timeout }
var
    Regs:       registers;
    PAbort:     boolean;
    Chk:        byte;
    TimeOut:    word;
begin
    if Length(S) = 0 then begin
        WritePrt: = True;
        Exit;
    end;
    PAbort: = ErrorCheck(False);
    while ((Length(S)0) and (not PAbort)) do begin
        if Keypressed then begin
            Regs.Ax: = $0000;     { read the keyboard }
            Intr($16,Regs);
            if Lo(Regs.Ax) = $00 then Chk: = 128 + Hi(Regs.Ax)
            else Chk: = Lo(Regs.Ax);
            if Chk = ESC then PAbort: = GetBool('Print cancel requested, Ok to stop?');
        end;
        if not PAbort then begin
            Regs.Dx: = $0000;          { select printer 1 }
            Regs.Ax: = Ord(S[1]);      { output 1 character }
            Intr($17,Regs);
            Timeout: = 0;
            while (((Hi(Regs.Ax) and 128) = 0) and (TimeOutWait)) do begin
                Inc(Timeout);
                Delay(1);
                Regs.Dx: = $0000;      { select printer 1 }
                Regs.Ax: = $0200;      { request printer status }
                Intr($17,Regs);
            end;
            if Timeout = PWait then PAbort: = (not PrinterReady);
            if not PAbort then Delete(S,1,1);
        end;
    end;
    while Keypressed do begin
        Regs.Ax: = $0000;     { clear the keyboard, just in case }
        Intr($16,Regs);
    end;
    WritePrt: = (not PAbort);
end;
```

Figure 3-5. Printer output function

developed to bypass the host computer's input/output system and write screen output directly to video memory. Screen output handled in this way results in the optimum display speed necessary for this research.

3. Printer output

Certain operations of the program require printed reports to collect and analyze simulation information. The standard Pascal language provides access to a printer with the WRITELN procedure. WRITELN presents a problem if error conditions occur during printer output. If, for example, the printer runs out of paper while printing, the error message returned by the printer will cause the display to scroll. No direct recovery of the correct screen display would be possible. It is also desirable to allow the user to interrupt printing at any time. The function shown in Figure 3-5 provides a solution to the problems stated above.

4. Error handling

Any computer program should adequately protect the user and the data from errors that may occur, either through system malfunction or incorrect entry of data. The software developed in this research addresses both types of errors.

```
procedure Int24On;
{ enable new Int24 error handler }
begin
    GetIntVec($24,OldInt24);     { save old Int24 vector }
    SetIntVec($24,@Int24);       { install new critical error handler }
    CritError:=0;                { and set global errors to zero }
    PasError:=0;
    AMSTError:=0;
end;


procedure Int24Off;
{ restore original Int24 error handler }
begin
    SetIntVec($24,OldInt24);     { restore old Int24 vector }
end;
```

Figure 3-6. Routines to enable and disable error handler

```
procedure Int24(Flags,CS,IP,AX,BX,CX,DX,SI,DI,DS,ES,BP:word);interrupt;
{ general purpose critical error handler }
type
    ScrPtr   =   ^ScrBuf;
    ScrBuf   =   array [1..320] of byte;
var
    Display, OldLine:    ScrPtr;
    AH,AL,OldAttr:       byte;
    Row,Col:             integer;
    Action:              char;
    ErrMsg:              string;
    ErrCode:             word;
    Ch:                  shortint;
    DevAttr:             ^word;
    DevName:             ^char;
begin
    ErrCode:=IOResultPrim;    { call IOResult before to clear }
    if IsMono then Display:=ptr($B000,Pred(MSGLINE)*160)      { save screen }
    else Display:=ptr($B800,Pred(MSGLINE)*160);
    New(OldLine);       OldLine^:=Display^;
    AH:=Hi(AX);         AL:=Lo(AX);
    Col:=WhereX;        Row:=WhereY;
    OldAttr:=TextAttr;  ErrMsg:='';
    if (AH and $80) = 0 then begin
        ErrCode:=Lo(DI);        ErrMsg:='DOS Critical Error';
    end
    else begin
        DevAttr:= Ptr(BP, SI+4);   { point to device attribute word }
        if (DevAttr^ and $8000)  0 then begin      { if bit 15 is on }
            Ch:=0;
            repeat
                DevName:=Ptr(BP,SI+$0A+Ch);   ErrMsg:=ErrMsg+DevName^;
                Inc(Ch);
            until (DevName^ =Chr(0)) or (Ch7);
            ErrMsg:=ErrMsg + ' not responding';    ErrCode:=$02;
        end
        else begin
            ErrMsg:='Bad File Allocation Table';    ErrCode:=$0D;
        end;
    end;
    GotoXY(1,MSGLINE);        TextAttr:=ErrorC;
    ClrEol; Write(' ',ErrMsg,' -- A)bort or R)etry?');      Beep;
    repeat  Action:=Upcase(Readkey);       until Action in [#27,'A','R'];
    Display^:=OldLine^;       Dispose(OldLine);
    GotoXY(Col,Row);          TextAttr:=OldAttr;
    case Action of
        #27,'A':    begin   CritError:=ErrCode;   AX:=0;    end;
        'R':        begin   CritError:=0;         AX:=1;    end;
    end;
    ErrCode:=IOResultPrim;    { call IOResult after to clear }
end;
```

Figure 3-7. Replacement interrupt 24 handler

MS-DOS computers automatically generate a class of errors called "critical errors" when certain error conditions are present. Under normal program operation, critical errors display an "Abort, Retry, Ignore" message on the screen and cause the screen to scroll. Critical errors are generated through the internal software interrupt number 24 hex. To prevent the screen scrolling, the software must replace the default interrupt 24 handler. Figure 3-6 shows the routines used to enable and disable the new interrupt 24 critical error handler. Figure 3-7 shows the replacement interrupt 24 handler used in this research.

The foundation for the simulation software developed in this research has been presented. The next section presents a discussion of the methods used to manipulate the classes in the simulation program.

E. Class Manipulation

The previous section presented the basic building blocks of the software created in this research. This section presents a discussion of the concept of classes as used in the context of object-oriented programming.

Classes allow implementation of the basic principles of object-oriented programming:

- Information hiding
- Data abstraction
- Dynamic binding
- Inheritance

Information hiding is implemented in the software by placing the declarations for variables inside Pascal units for each class. Each class is only aware

of its own format. The objects within a class unit cannot be directly accessed by objects of another class.

Data abstraction is implemented by creating generic class types in the Pascal language. These class types are generalized templates that contain sufficient information for self-definition. The class types are known globally only as place-holders in memory. All support routines in the program are written to manipulate these generic class types.

Dynamic binding is achieved by avoiding direct manipulation of fields within objects whenever possible. The message handling system described later performs the manipulation of the actual data within objects of a class. Messages may change during operation of the program and are not reliant on the compilation of the program source code.

Inheritance is a direct result of the combination of class types and Pascal units. When objects of a particular class are created, they automatically assume the structure of the parent class. Procedures that act on that class type are also automatically inherited. This research only allows single-level inheritance mechanisms. The following sections provide additional details of the structure of classes in this research and the methods used for class manipulation.

1. Class type

The data abstraction principle of object-oriented programming dictates that classes of objects should be defined in such a manner that the class knows of its own structure but all classes share the same basic construction. In a strongly-typed procedural language such as Pascal, this abstraction of classes is accomplished through the use of a global class type definition. In this research, the class type is constructed on a field by field basis with the record defininition shown in Figure 3-8.

```
DBFieldPtr =  ^DBField;

DBField    =  record  { input screen field definition record }
                 Title  :  DBTitleStr;    { field title }
                 FType  :  char;          { field type }
                 Len    :  byte;          { field length }
                 Decs   :  byte;          { decimal precision }
                 X      :  byte;          { X position on screen }
                 Y      :  byte;          { Y position on screen }
                 Page   :  byte;          { field page on screen }
                 ALen   :  byte;          { byte length of field }
                 AOfs   :  integer;       { offset into record }
                 CCase  :  char;          { up/low conversion type }
                 Mand   :  boolean;       { mandatory entry? }
                 Calc   :  boolean;       { calculated field }
                 KType  :  char;          { key: N)o D)ups U)nique }
                 OkSet  :  MenuSet;       { allowable entry chars }
                 Form   :  DBFormStr;     { formula for this field }
                 WTitle :  boolean;       { on screen w/title? }
               end;
```

Figure 3-8. Class field definition record type

The DBField record type described in Figure 3-8 is key to the object-oriented nature of this research. The information used in this record type is used to construct all classes. Note that sufficient information is available for display and modification of objects created with this record structure. The fields within the DBField record closely parallel typical definitions for object classes under the object-oriented paradigm. A combination of DBField records can be used to define a class. After a class is defined, generic class manipulation methods can be used on the class without knowing the exact structure of the class.

2. Class creation

A programmer using the methods developed in this research can quickly create new class types by using the DBField record type to define the individual fields in a class. The simulation program developed for this research used a separate program to define the classes. The separate program allowed for on-screen editing

```
procedure DBPutFieldDef(  var     DFT:DBField;        Title:DBTitleStr;
                                  FType:char;         Len,Decs,X,Y,Page,ALen:byte;
                                  AOfs:integer        CCase:char;
                                  Mand,Calc:boolean;
                                  KeyTyp:char;        OkSet:MenuSet;
                                  Form:DBFormStr;     WTitle:boolean);
{ put a definition into a DBField }
begin
    DFT.Title:=Title;       DFT.FType:=FType;    DFT.Len:=Len;
    DFT.Decs:=Decs;         DFT.X:=X;            DFT.Y:=Y;
    DFT.Page:=Page;         DFT.ALen:=ALen;      DFT.AOfs:=AOfs;
    DFT.CCase:=CCase;       DFT.Mand:=Mand;      DFT.Calc:=Calc;
    DFT.KType:=KeyTyp;      DFT.OkSet:=OkSet;    DFT.Form:=Form;
    DFT.WTitle:=WTitle;
end;
```

Figure 3-9. Procedure to place class definition into memory

```
function DBLoadDef(FName:string;  ObjBuffer,ObjTBuffer,ObjBBuffer:DBBufPtr;
          ObjF:DBFieldArray;      ObjScreen:WindowPtr):boolean;
{ load database definition }
var
    I,NFlds:byte;     DBN:integer;     SStr:string[DBMaxFldLen];
    DDFR:DBFileRec;  DDFV:file of DBFileRec;
begin
    DBLoadDef:=False;          NFlds:=0;  FillChar(SStr,Succ(DBMaxFldLen),#32);
    ObjScreen^.ULX:=1;         ObjScreen^.ULY:=Pred(DBMINY);
    ObjScreen^.LRX:=80;        ObjScreen^.LRY:=Succ(DBMAXY);
    Assign(DDFV,FName);        Reset(DDFV);
    FillChar(ObjBuffer^,Succ(DBMAXRECLEN),0);
    FillChar(ObjTBuffer^,Succ(DBMAXRECLEN),0);
    while not EOF(DDFV) do begin
        Read(DDFV,DDFR);    case DDFR.RType of
        0:  begin    { field definition }
            Inc(NFlds); Move(DDFR.FieldDef.Title,ObjF[NFlds]^,SizeOf(DBField));
            ObjF[NFlds]^.Title:=PadRight(ObjF[NFlds]^.Title,' ',DBTITLELEN);
            case ObjF[NFlds]^.FType of
                'A': begin    SStr[0]:=Chr(ObjF[NFlds]^.Len);
                     DBPutBuffer(SStr,ObjBuffer,ObjF[NFlds]^);end;
                'E': DBPutBuffer(DBBENTRY,ObjBuffer,ObjF[NFlds]^);end;
            end;
        1:  begin          { screen line }
            for I:=0 to 79 do case Hi(DDFR.ScrLine.Cont[I]) of
                1:  DDFR.ScrLine.Cont[I]:=Lo(DDFR.ScrLine.Cont[I])+(LowC shl 8);
                2:  DDFR.ScrLine.Cont[I]:=Lo(DDFR.ScrLine.Cont[I])+(NormC shl 8);
                3:  DDFR.ScrLine.Cont[I]:=Lo(DDFR.ScrLine.Cont[I])+(InvC shl 8);
            end;
            Move(DDFR.ScrLine.Cont,ObjScreen^.Add[DDFR.ScrLine.Line],160);
        end;    end;
    end;
    Close(DDFV);
    for I:=Succ(NFlds) to DBMAXFIELDS do ObjF[I]^:=ObjF[0]^;
    Move(ObjBuffer^,ObjBBuffer^,Succ(DBMAXRECLEN));  DBLoadDef:=TRUE;
end;
```

Figure 3-10. Procedure to load class definitions

```
procedure ObjectImit(    ObjNum:byte;
                         var ObjScreen:WindowPtr;
                         var ObjBuffer,ObjTBuffer,ObjBBuffer:DBBufPtr;
                         var ObjF:DBFieldArray);
{ initialize memory for use by an object class definition }
var
    I:          integer;
    NFlds:      byte;
    MemOk:      boolean;
begin
    MemOk:=True;
    I:=0;
    if MaxAvailSizeOf(WindowArray)+MinMem then
        GetMem(ObjScreen,SizeOf(WindowArray))
    else
        MemOk:=False;
    if MemOk then
        MemOk:=DBGetWorkingBuffers(ObjBuffer,ObjTBuffer,ObjBBuffer);
    if MemOk then begin
        I:=0;
        while ((IAXFIELDS) and (MemOk)) do begin
            if MaxAvailSizeOf(DBField)+MinMem then
                GetMem(ObjF[I],SizeOf(DBField))
            else MemOk:=False;
            Inc(I);
        end;
    end;
    if not MemOk then begin
        Msg('Insufficient memory to run program');
        Halt;
    end;
    with ObjF[0]^ do begin
        Title   := CharStr(' ',10);
        FType   := DBBCHAR;
        Len     := DBBBYTE;
        Decs    := DBBBYTE;
        X       := DBBBYTE;
        Y       := DBBBYTE;
        Page    := DBBBYTE;
        ALen    := DBBBYTE;
        AOfs    := DBBINT;
        CCase   := DBUPLOW;
        Mand    := DBNMAND;
        Calc    := DBNCALC;
        KType   := DBNKEY;
        OkSet   := [];
        Form    := DBBFORM;
    end;
    if not   DBLoadDef(   DBMakeName(CLSNAMES[ObjNum],0,0),
                          ObjBuffer,
                          ObjTBuffer,
                          ObjBBuffer,
                          ObjF,ObjScreen)
        then Halt;
end;
```

Figure 3-11. Procedure to intialize classes

of the DBField parameters and then saved the DBField records in a disk file. Separate editing of the DBField parameters allows for data abstraction in object-oriented programming.

The simulation program must only load the DBField records from a disk file for each class when the program is started. The procedure shown in Figure 3-9 is used to place the DBField records into memory where they can later be used by the classes as described later.

A disk file exists for each class in the simulation program. The procedure shown in Figure 3-10 is called once for each class in the program to load the definition into memory. The procedure shown in Figure 3-11 is used to initialize the class for use in the simulation program. After the class definitions have been placed in memory, the simulation program has sufficient information for class manipulation in a generic fashion.

3. Mapping classes to object types

The class initialization routines shown in Figure 3-11 create a space in memory for a class definition. This memory space is treated in a generic fashion by the simulation program. The object-oriented paradigm mandates that the individual objects within a class must be aware of their own structure and data contents. This awareness is accomplished by mapping the generic class definition to a specific Pascal record type within each class unit. By restricting the specific record definition of a class to the unit that contains the class methods, information hiding is maintained.

Mapping of class definitions to objects is achieved through the use of pointers in Pascal. A pointer is a memory address. The simulation program must only be aware of the address of the current working object. Each class unit contains a

memory buffer used to hold the current working object. This memory buffer is maintained in a fixed and known location. A Pascal record type may then be defined within each class unit. The working object is then transferred to the fixed-location buffer whenever the object must be manipulated. The procedures shown in Figure 3-12 are used to move the current object in a class to the working buffer. The procedures and functions described in the next section may then be used to manipulate the objects within a class.

Within each class unit, several pointers are maintained to assist in locating a specific object when manipulation of the object is required. All objects of a particular class are collected in a linked list. The common factors within object definition records are pointers to the next and previous instances of an object. Pointers to the first, last, and current working object are also maintained within each class. Initially, the first, last, and current object pointers are set to the nil memory address which points to nothing and indicates an empty list.

Note how the procedures shown in Figure 3-12 use the information contained in the generic class definitions to determine the size and location of fields within an object. These procedures allow a field within an object to be directly accessed and modified. The actual layout of the data fields in an object are only known within a

```
procedure DBGetBuffer(var FData; ObjBuffer:DBBufPtr; DFT:DBField);
{ get contents of buffer at defined field }
begin
    Move(ObjBuffer^[DFT.AOfs],FData,DFT.ALen);
end;


procedure DBPutBuffer(var FData; ObjBuffer:DBBufPtr; DFT:DBField);
{ put contents into buffer }
begin
    Move(FData,ObjBuffer^[DFT.AOfs],DFT.ALen);
end;
```

Figure 3-12. Procedures to load and save object buffers

class unit and cannot be externally modified, thus maintaining the information hiding principle of the object-orient programming paradigm.

The most similar action in SLAM to create a new class would be the creation of a new type of network node. To create a new network node, a programmer would have to write the supporting code for the new node. Next, the programmer would have to modify other code segments in SLAM that would potentially reference the new network node. All data interdependencies at the source code level must be examined and possibly modified. The entire process could potentially take a great deal of time and resources. The comparative complexity of new class creation in the software created for this research is minor because data interdependencies between classes do not exist in keeping with the object-oriented programming paradigm.

4. Object creation and manipulation

Many methods are common between classes. Methods are required to create and manipulate specific instances of an object. Given the basic building blocks for class definition and access described previously, new instances of an object can be created and existing instances of an object can be accessed and manipulated.

The procedures shown in Figure 3-13 and Figure 3-14 demonstrate object creation and deletion. Note that knowledge of the internal structure of a class is not required for these operations. This intentional ignorance provides a high level of modularity to the program. Class methods remain relatively simple and portable between class units.

Note the use of the PutObjInBuffer method shown in Figure 3-15 when deleting an object through the DeleteCurrObject method. The PutObjInBuffer method places an object into the class working buffer discussed previously. After an object has been placed in the working buffer, it can be accessed and manipulated

```
function GetNewObject:boolean;
{ allocate a new object instance and add to end of linked list }
begin
    GetNewObject:=False;
    if MaxAvail < SizeOf(ObjRec)+MinMem then
        Exit;
    GetMem(TPtr,SizeOf(ObjRec));
    TPtr^.Prev:=LastObj;
    TPtr^.Next:=nil;
    if TPtr^.Prev <> nil then
        TPtr^.Prev^.Next:=TPtr;
    CurrObj:=TPtr;
    LastObj:=TPtr;
    if FirstObj=nil then
        FirstObj:=TPtr;
    Move(ObjBBuffer^,CurrObj^,ObjSize);
    GetNewObject:=True;
end;
```

Figure 3-13. Method for object instance creation

```
function DeleteCurrObject:boolean;
begin
    DeleteCurrObject:=False;
    if CurrObj=nil then Exit;
    TPtr:=CurrObj;
    if FirstObj=TPtr then
        FirstObj:=FirstObj^.Next;
    if LastObj=TPtr then
        LastObj:=LastObj^.Prev;
    if CurrObj^.Prev <> nil then
        CurrObj:=CurrObj^.Prev
    else if CurrObj^.Next <> nil then
        CurrObj:=CurrObj^.Next
    else
        CurrObj:=nil;
    if TPtr^.Prev <> nil then
        TPtr^.Prev^.Next:=TPtr^.Next;
    if TPtr^.Next <> nil then
        TPtr^.Next^.Prev:=TPtr^.Prev;
    Dispose(TPtr);
    PutObjInBuffer;
    DeleteCurrObject:=True;
end;
```

Figure 3-14. Method for object deletion

with the methods shown in Figures 3-16 through 3-23. Note throughout these methods that specific data fields within the class definitions are never referenced. Figure 3-16 presents the method used to display the current object on the computer screen.

Figure 3-17 shows the method used to clear data from the current object. A key value of "BLANK" is placed in the formula field of a class definition if the field is to be cleared when this method is invoked. The method shown in Figure 3-18 is

```
procedure PutObjInBuffer;
{ put the Current object in the display buffer }
begin
    if CurrObj < > nil then Move(CurrObj^,ObjBuffer^,ObjSize)
    else Move(ObjBBuffer^,ObjBuffer^,ObjSize);
end;
```

Figure 3-15. Method to place object in working buffer

```
procedure ShowObject;
{ show current object }
var
    FData:      DBFDataArray;
    FldNum:     byte;
begin
    if CurrClsObjNum then begin
        if (not SStep) then Exit;
        CurrCls: = ObjNum;
    end;
    if ((not SStep) and (CurrObjLastDisp) and (not Paused)) then Exit;
    RestoreWindow(ObjScreen^);
    FldNum: = 1;
    while ObjF[FldNum]^.Page = 1 do begin
        DBGetBuffer(FData,ObjBuffer,ObjF[FldNum]^);
        with ObjF[FldNum]^ do
            WriteFast(X,Y,InvC,MakeStr(FData,Len,Decs,FType));
        Inc(FldNum);
    end;
    LastDisp: = CurrObj;
end;
```

Figure 3-16. Method to display an object

invoked if all objects of a class are to be cleared. This method traverses the linked list of objects and calls the method to clear a single object.

The methods shown in Figure 3-19 demonstrate how the linked list of objects is traversed to select either the previous instance or the next instance of a particular object. These methods are used by higher level routines discussed later.

Many of the instances of an object created during the execution of the program require the user to enter data in the data fields. The object-oriented paradigm suggests that access to the data fields of an object should be accomplished without direct knowledge of the format of that data. The software created in this research performs this data entry task in much the same fashion as the object

```
procedure ClearCurrObject;
{ clear data from object }
var
    FData:      DBFDataArray;
    FldNum:     byte;
begin
    FldNum:=1;
    while ObjF[FldNum]^.Page=1 do begin
        if StripLeft(StripRight(ObjF[FldNum]^.Form,' '),' ')='BLANK' then begin
            DBGetBuffer(FData,ObjBBuffer,ObjF[FldNum]^);
            DBPutBuffer(FData,ObjBuffer,ObjF[FldNum]^);
        end;
        Inc(FldNum);
    end;
end;
```

Figure 3-17. Method to clear an object

```
procedure ClearAllObjects;
{ clear data from all objects }
begin
    TPtr:=FirstObj;
    while TPtr <> nil do begin
        ClearCurrObject;
        TPtr:=TPtr^.Next;
    end;
end;
```

Figure 3-18. Method to clear all instances of an object

```
function GetNextObject:boolean;
{ get the next object }
begin
    GetNextObject: = False;
    if CurrObj = nil then Exit;
    if CurrObj ^ .Next = nil then Exit;
    CurrObj: = CurrObj ^ .Next;
    PutObjInBuffer;
    GetNextObject: = True;
end;


function GetPrevObject:boolean;
{ get the previous object }
begin
    GetPrevObject: = False;
    if CurrObj = nil then Exit;
    if CurrObj ^ .Prev = nil then Exit;
    CurrObj: = CurrObj ^ .Prev;
    PutObjInBuffer;
    GetPrevObject: = True;
end;
```

Figure 3-19. Methods to get the next or previous object

creation. The program knows the layout of the objects internally to the class units, but access to the specific fields is accomplished in a generic way through the class definitions described previously.

The method shown in Figure 3-20 demonstrates user data entry that conforms to the object-oriented paradigm. Note how methods defined previously are used to access individual data fields within the object. As each field is accessed, the data corresponding to that field is moved to a temporary buffer, manipulated according to the class definition, then moved back to the object.

The methods shown in Figures 3-21, 3-22, and 3-23 are used to print the contents of all current instances of an object, load all instances of an object from a disk file, and save all instances of an object to a disk file.

```
procedure GetObject(RType:byte);
{ enter or update data in an object instance }
var
    FData:      DBFDataArray;
    Fin:        boolean;
    FldNum:     byte;
    FFld:       byte;
    Next:       byte;
begin
    if ((RType=1) and (CurrObj=nil)) then Exit;
    Next:=CR;
    FldNum:=1;
    Fln:=True;
    while ObjF[FldNum]^.Calc do Inc(FldNum);
    FFld:=FldNum;
    ShowMenu(RType+125);
    repeat
        Fin:=False;
        Move(ObjBuffer^,ObjTBuffer^,ObjSize);    { save current object }
        if RType = 2 then begin
            Move(ObjBBuffer^,ObjBuffer^,ObjSize);    { new blank record }
            if not GetNewObject then Next:=ESC;    { allocate new object }
        end;
        FldNum:=FFld;
        ShowObject;
        if Next <> ESC then repeat
            DBGetBuffer(FData,ObjBuffer,ObjF[FldNum]^);
            DBGetField(FData,Next,ObjF[FldNum]^,RType,InvC,EMPTYSET);
            DBPutBuffer(FData,ObjBuffer,ObjF[FldNum]^);
            DBGetNextField(FldNum,Next,ObjF);
        until Next in [ESC,F5,F6,F10];
        Fin:=(Next in [ESC,F10]);
        case Next of
            ESC:    begin    { abort }
                        Move(ObjTBuffer^,ObjBuffer^,ObjSize);
                        if RType=2 then if DeleteCurrObject then ;{ delete object }
                    end;
            F5:     begin    { previous object }
                        Move(ObjBuffer^,CurrObj^,ObjSize);
                        if not GetPrevObject then ;
                    end;
            F6:     begin    { next object }
                        Move(ObjBuffer^,CurrObj^,ObjSize);
                        if RType=1 then if not GetNextObject then ;
                    end;
            F10:    Move(ObjBuffer^,CurrObj^,ObjSize);
        end;
        ShowObject;
    until Fin;
    ShowMenu(CmdList);
    if HilightCommand(0) then ;
end;
```

Figure 3-20. Method to allow user entry of data in an object

```
procedure ReportSimulation;
{ print all object detail }
var
    FData:      DBFDataArray;      FldNum:     byte;
begin
    CurrObj:=FirstObj;      if CurrObj=nil then Exit;
    if not PrinterReady then Exit;
    while CurrObj <> nil do begin
        PutObjInBuffer;      FldNum:=1;
        while ObjF[FldNum]^.Page=1 do begin
            DBGetBuffer(FData,ObjBuffer,ObjF[FldNum]^);
            with ObjF[FldNum]^ do
                if not WritePrt(Title+': '+
                    MakeStr(FData,Len,Decs,FType)+PCRLF+PCRLF) then Exit;
            Inc(FldNum);
        end;
        if not WritePrt(PFF) then Exit;
        CurrObj:=CurrObj^.Next;
    end;
end;
```

Figure 3-21. Method to print contents of objects

```
procedure LoadObjects;
{ load simulation objects from disk }
var
    TObj:   ObjRec;      ObF:    file of ObjRec;
begin
    while DeleteCurrObject do ;        { delete current objects from memory }
    if not FileExist(DBMakeName(SimName,1,ObjNum)) then Exit;
    Assign(ObF,DBMakeName(SimName,1,ObjNum));      Reset(ObF);
    while (not EOF(ObF)) do begin
        Read(ObF,TObj);
        if not GetNewObject then begin
            Close(ObF);
            Msg('Insufficient memory to load simulation, program halted'); Halt;
        end;
        Move(TObj,CurrObj^,ObjSize);
    end;
    Close(ObF);          CurrObj:=FirstObj;
    PutObjInBuffer;      ShowObject;
end;
```

Figure 3-22. Method to load objects from a disk file

```
procedure SaveObjects;
{ save simulation objects to disk }
var
    ObF:    file of ObjRec;
begin
    { save objects to disk file }
    TPtr:=FirstObj;
    Assign(ObF,DBMakeName(SimName,1,ObjNum));
    Rewrite(ObF);
    while TPtr <> nil do begin
        Write(ObF,TPtr^);
        TPtr:=TPtr^.Next;
    end;
    Close(ObF);
end;
```

Figure 3-23. Method to save objects to a disk file

The methods described in this section are all used in a generic way to manipulate classes and objects. It is important to recognize that these routines do not rely on any particular format of the classes. These routines strictly follow the object-oriented philosophy as discussed previously. Generic treatment of classes and objects allows methods to be created that will correctly function regardless of the structure of the target class. Code portability and programmer efficiency is enhanced and chances of programmer error are reduced.

The software created in this research now has general object-oriented capabilities. The primary goal of this research is to develop object-oriented simulation capabilities in a strongly-typed procedural language. The general routines described thus far are used as building blocks for the next phase of the software development. The next section provides a description of the object-oriented simulation capabilities created in this research.

F. Object-Oriented Simulation Facilities

The methods discussed in the previous section treat classes and objects in a generic way. The primary goal of this research is to integrate object-oriented simulation capabilities into a strongly-typed procedural language. This section describes the object-oriented facilities developed in direct support of discrete-event simulation.

The first step toward object-oriented simulation is to define the classes necessary for discrete-event simulation. Next, the basis for message handling is presented. The current and future events calendar responsible for control of the simulation is then described followed by an overview of the simulation clock. Finally, the messages used in discrete-event simulation are presented.

1. Simulation classes

There are four primary classes that must be incorporated into the simulation program to support the desired simulation environment. They are the "Simulation" class, the "Entity" class, the "Routing" class, and the "Server/Queue" class.

In keeping with the object-oriented paradigm, the data that are relevant to each of these classes are maintained within the associated instance variables. It is important to note that the use of classes as a generic representation of simulation objects allows the generation of multiple instances of an object. A complex simulation model can then be built by using instances of an object without regard to the interaction between these objects which is automatic.

Figures 3-24 through 3-27 show the internal structure of each of the simulation classes. Note that these record structures are only known within the class and therefore follow the information hiding construct of object-oriented programming.

```
ObjRec     =  record  { simulation object record }
                 Status:          longint;
                 Instance:        InstType;
                 Desc:            string[25];
                 MaxTime:         real;
                 CurrTime:        real;
                 CurrQty:         real;
                 MinTInSys:       real;
                 MaxTInSys:       real;
                 AvgTInSys:       real;
                 Next:            ObjRecPtr;
                 Prev:            ObjRecPtr;
               end;
```

Figure 3-24. Simulation class definition

```
ObjRec     =  record  { entity object record }
                 Status:          longint;
                 Instance:        InstType;
                 TypeCode:        real;
                 CurrLoc:         InstType;
                 CreateTime:      real;
                 StartTime:       real;
                 TimeInSys:       real;
                 WillFail:        boolean;
                 Next:            ObjRecPtr;
                 Prev:            ObjRecPtr;
               end;
```

Figure 3-25. Entity class definition

```
ObjRec     =  record  { routing object record }
                 Status:          longint;
                 Instance:        InstType;
                 Desc:            string[25];
                 EntType:         real;
                 CurrLoc:         InstType;
                 Dist:            InstType;
                 Mean:            real;
                 Std:             real;
                 FailPerc:        real;
                 FailTo:          InstType;
                 NextLoc:         InstType;
                 BalkLoc:         InstType;
                 Next:            ObjRecPtr;
                 Prev:            ObjRecPtr;
               end;
```

Figure 3-26. Routing class definition

```
ObjRec     =  record  { server/queue object record }
                 Status:          longint;
                 Instance:        InstType;
                 Desc:            string[25];
                 Capacity:        real;
                 SrvStatus:       StatusType;
                 CurrQty:         real;
                 MaxQty:          real;
                 AvgQty:          real;
                 TotalQty:        real;
                 Utilized:        real;
                 MinTBA:          real;
                 MaxTBA:          real;
                 MeanTBA:         real;
                 MinTime:         real;
                 MaxTime:         real;
                 MeanTime:        real;
                 LastArrival:     real;
                 Next:            ObjRecPtr;
                 Prev:            ObjRecPtr;
              end;
```

Figure 3-27. Server/queue class definition

The data contained within each object is used to track key simulation parameters during the execution of the simulation program. Object data are constantly presented to the user during program execution. Some of the data are initially entered by the user of the program while other data are maintained by the program. In all cases, no object can directly access data in another object. All interaction between objects is performed through messages passed between objects.

2. Message handling

The foundation of the object-oriented simulation program created in this research is the message handling system. Messages are the only form of communication between objects. Figure 3-28 shows the format of messages in the simulation program. Every message in the program follows the standard message format, although some of the fields in the message packet may not be used.

```
MsgPacketType =  record
        FromCls:    byte;          { from which class }
        FromInst:   InstType;      { from which instance }
        ToCls:      byte;          { to which class }
        ToInst:     InstType;      { to which instance }
        Message:    MsgType;       { the actual message }
        Number:     real;          { a number to pass in message }
        Clock:      real;          { time to execute message (-1.0 = immediate) }
        Next:       MsgPacketPtr;  { pointer to next message }
    end;
```

Figure 3-28. Message packet format

The globally accessible procedure shown in Figure 3-29 is used to place messages on the message queue. Messages are placed in the message queue according to the time entered in the clock field of the message. The messages are automatically stored in time sorted order. Messages are then taken one at a time from the head of the message queue.

The use of an ordered list for the message queue directly corresponds to the events calendars found in discrete-event simulation. Strict adherence to the time-ordered structure of the message queue ensures that messages in the simulation system will follow the time-ordering necessary for simulation synchronization. The use of object-oriented message passing effectively removes the necessity for the traditional current and future events calendars.

Figure 3-30 shows the main routine used to control the simulation program. The main program commands are implemented in this routine. The main loop of this routine checks for user input and acts on that input if found. If no user input is pending, control is passed to the message checking routine shown in Figure 3-31. This routine is aware of the different classes and uses the message packet parameters to determine where the message should be sent. Note that this routine does not require knowledge regarding the internal structure of the classes.

```
procedure SendMsg(FromCls:byte;FromInst:InstType; ToCls:byte; ToInst:InstType;
          Message:MsgType; Number,Clock:real);
{ send a Message to/from the indicated Class, Instance, optionally with Number }
var
    MsgPacket:      MsgPacketPtr;
    TPtr,Lptr:      MsgPacketPtr;
    Done:           boolean;
    MsgNum:         byte;
begin
    if MaxAvail<=SizeOf(MsgPacketType)+MinMem then begin
        Msg('Insufficient memory for message queue, program aborted'); Halt;
    end;
    GetMem(MsgPacket,SizeOf(MsgPacketType));    { allocate message memory }
    MsgPacket^.FromCls:=FromCls;    { assign from class }
    MsgPacket^.FromInst:=FromInst;  { assign from instance }
    MsgPacket^.ToCls:=ToCls;        { assign to class }
    MsgPacket^.ToInst:=ToInst;      { assign to instance }
    MsgPacket^.Message:=Message;    { assign message }
    MsgPacket^.Number:=Number;      { assign number }
    MsgPacket^.Clock:=Clock;        { assign clock time }
    if SStep then begin             { display message }
        MsgNum:=Ord(MsgPacket^.Message);
        WriteMsg(NormC,'Send:'+ClsNames[MsgPacket^.FromCls]+
                ','+MsgPacket^.FromInst+
                'to'+ClsNames[MsgPacket^.ToCls]+','+MsgPacket^.ToInst+
                '"'+SoopMsgs[MsgNum]+'"'+
                MakeStr(MsgPacket^.Number,0,2,'R')+
                ','+MakeStr(MsgPacket^.Clock,0,2,'R'));
        if GetAKey0 then ;
    end;
    Inc(MsgCount);
    WriteAt(60,1,CHead+MakeStr(MsgCount,5,0,'W'));
    MsgPacket^.Next:=FirstMsg;      { start as new first message }
    if FirstMsg=nil then begin      { this is the only message }
        FirstMsg:=MsgPacket;  Exit;
    end;
    if MsgPacket^.Clock<FirstMsg^.Clock then begin    { belongs first }
        MsgPacket^.Next:=FirstMsg;
        FirstMsg:=MsgPacket;
        Exit;
    end;
    Done:=False;
    TPtr:=FirstMsg;                 { point to first message }
    LPtr:=TPtr;
    while (not Done) do begin       { find appropriate position }
        MsgPacket^.Next:=TPtr^.Next;
        TPtr^.Next:=MsgPacket;
        if TPtrFirstMsg then LPtr^.Next:=TPtr;
        Done:=(MsgPacket^.Next=nil);
        if not Done then begin
            LPtr:=TPtr;
            TPtr:=MsgPacket^.Next;
            Done:=(MsgPacket^.Clock>TPtr^.Clock);
        end;
    end;
end;
```

Figure 3-29. Message sending procedure

```
procedure MessageHandler;
{ main program message handler }
var
    Ch:     byte;          { working character variable }
    M:      longint;       { temporary memory check variable }
begin
    CurrCls:=1;            { initialize currently diplayed class }
    Paused:=True;          { current simulation is paused }
    SStep:=False;          { single step is off }
    MsgCount:=0;           { message count is zero }
    SimClock:=0.0;         { set to a new simulation }
    FirstMsg:=nil;         { clear the message queue }
    WriteAt(1,1,CHead+
        'SIMULATION WITH OBJECT-ORIENTED PROGRAMMING '+
        +'Msg Count    :Sim:');
    SimName:='    ';       { no current simulation }
    ShowMenu(1);           { display menu }
    if HilightCommand(0) then ;      { hilite command list }
    SendMsg(MAILMAN,NINST,CurrCls,NINST,SHOW_CURR_OBJ,0.0,PRIORITY);
    repeat                 { now go into command loop }
        if CurrCommand <> 0 then begin        { if user command is pending, act on it }
        case CurrCommand of
    1:  SimulationClear;     { clear the data in the simulation objects }
    2:  SendMsg(MAILMAN,NINST,CurrCls,NINST,DELETE_OBJ,0.0,PRIORITY);
    3:  SendMsg(MAILMAN,NINST,CurrCls,NINST,ENTER_OBJ,0.0,PRIORITY);
    4:  SimulationLoad;      { Load simulation from disk }
    5:  SimulationOptions;   { set simulation options }
    6:  SimulationStartStop; { Proceed with or Pause current simulation }
    7:  SimulationReport;    { Report (print) simulation reports }
    8:  SimulationSave;      { Save simulation to disk }
    9:  SendMsg(MAILMAN,NINST,CurrCls,NINST,UPDATE_OBJ,0.0,PRIORITY);
    10: if GetBool('Are you sure you want to quit?') then Halt;      { Quit program }
        end;
        CurrCommand:=0;
    end     else if Keypressed then begin
            Ch:=KeyBoard(AllChar+
            [BACK,CR,ESC,LEFT,RIGHT,PGUP,PGDN,F5,F6,178],2);
        if Ch=ESC then Ch:=81;
        case Ch of
    F5:SendMsg(MAILMAN,NINST,CurrCls,NINST,SHOW_PREV_OBJ,0.0,PRIORITY);
    F6:SendMsg(MAILMAN,NINST,CurrCls,NINST,SHOW_NEXT_OBJ,0.0,PRIORITY);
    PGUP:begin     { show previous class and its current object instance }
        CurrCls:=Succ((CurrCls+MaxClasses-2) mod MaxClasses);
        SendMsg(MAILMAN,NINST,CurrCls,NINST,SHOW_CURR_OBJ,0.0,PRIORITY);
        end;
    PGDN:begin     { show next class and and its current object instance }
        CurrCls:=Succ(CurrCls mod MaxClasses);
        SendMsg(MAILMAN,NINST,CurrCls,NINST,SHOW_CURR_OBJ,0.0,PRIORITY);
        end;
    BACK,LEFT:       if HilightCommand(-1) then;
    SPACE,RIGHT:     if HilightCommand(1) then;
    13,33..47,58..126:     RunCommand(Ch);
        end;    { case }
    end     else CheckMessages;      { check the message queue }
    until False;                { never leave this loop! (program quits from HALT) }
end;
```

Figure 3-30. Main program loop

```
procedure CheckMessages;
{ check the message queue for pending messages }
var
    MData:      MsgPacketType;      { avoid pointer type to retain data }
    TPtr:       MsgPacketPtr;
    Done:       boolean;
    MsgNum:     byte;
begin
    Done:=(FirstMsg=nil);
    while not Done do begin
        if Keypressed then Exit;        { allows user to interrupt }
        if ((FirstMsg^.ClockPRIORITY) and (Paused)) then Exit; { simulation paused }
        if SStep then begin             { display message }
            MsgNum:=Ord(FirstMsg^.Message);
            WriteMsg(NormC,'Recv: '+
                ClsNames[FirstMsg^.FromCls]+','+FirstMsg^.FromInst+
                ' to '+ClsNames[FirstMsg^.ToCls]+','+FirstMsg^.ToInst+
                ' "'+SoopMsgs[MsgNum]+'" '+
                MakeStr(FirstMsg^.Number,0,2,'R')+','+
                MakeStr(FirstMsg^.Clock,0,2,'R'));
            if GetAKey=ESC then begin
                Paused:=True;
                ShowMenu(1);        { show the correct command list }
                if HilightCommand(0) then ;
                Exit;
            end;
        end;
        if FirstMsg^.ClockSimClock then begin       { update the simulation clock }
            SimClock:=FirstMsg^.Clock;
            SendMsg(MAILMAN,NINST,SIMULATE,NINST,
                        UPDATE_CLOCK,SimClock,PRIORITY);
        end;
        MData:=FirstMsg^;       { get message from front of message queue }
        TPtr:=FirstMsg;         { delete the message & reset pointers }
        FirstMsg:=FirstMsg^.Next;
        Dispose(TPtr);
        Dec(MsgCount);
        WriteAt(60,1,CHead+MakeStr(MsgCount,5,0,'W'));
        case MData.ToCls of     { send message to appropriate place }
            MAILMAN:    case MData.Message of   { message to mailman, handle it here }
                END_SIMULATION : begin      { end current simulation }
                    Msg('Simulation completed');
                    Paused:=True;
                    ShowMenu(1);
                    if HilightCommand(0) then ;
                end;
            end;
            SIMULATE:   SimClass(MData);
            ENTITY:     EntClass(MData);
            ROUTING:    RteClass(MData);
            SERVQUE:    SrvClass(MData);
        end;
        Done:=(FirstMsg=nil);
        if not Done then Done:=(FirstMsg^.Clock>PRIORITY); { no priority messages }
    end;
end;
```

Figure 3-31. Message handler (MAILMAN)

```
MsgType    =  (
        NMSG,                { nil message }
        CLEAR_OBJ,           { clear data fields in objects of a class }
        DELETE_OBJ,          { delete an instance of an object }
        ENTER_OBJ,           { enter (user) new data for an object }
        LOAD_OBJ,            { load simulation objects from disk }
        SAVE_OBJ,            { save simulation objects to disk }
        SHOW_CURR_OBJ,       { show current instance of an object }
        SHOW_NEXT_OBJ,       { show next instance of an object }
        SHOW_PREV_OBJ,       { show previous instance of an object }
        UPDATE_OBJ,          { update (user) the data for an object }
        UPDATE_CLOCK,        { update the simulation clock }
        GEN_ARR_TIME,        { determine which arrival to generate & when }
        GEN_ARRIVAL,         { general next arrival of an entity }
        GET_NEXT_RTE,        { get next routing for an object }
        GET_ALT_RTE,         { request for service/queue denied, get alternate route }
        GET_FAIL_RTE,        { request for service/queue after failure }
        GET_FAIL_RTRY,       { request for service/queue denied after failure, retry }
        REQ_SQ_ENTRY,        { entity request for entry to service or queue }
        REQ_SQ_GRANTED,      { request for service/queue granted }
        REQ_SQ_DENIED,       { request for service/queue denied }
        REQ_SQ_COMP,         { request completion of service time }
        SCH_SQ_COMP,         { schedule the completion of service }
        SQ_COMPLETE,         { a service has been completed }
        ENTITY_SQ_COMP,      { tell entity is has completed a service/queue }
        ENTITY_LEAVE_SQ,     { tell service/queue that entity has left }
        ENTITY_SET_FAIL,     { set an entity to fail service }
        ENTITY_NO_FAIL,      { set an entity not to fail service }
        ENTITY_DEP,          { entity has departed system }
        LEAVE_SYS,           { tell entity to leave system }
        REPORT_SIM,          { report on the simulation }
        END_SIMULATION       { end the current simulation }
    );
```

Figure 3-32. Messages used in discrete-event simulation

3. Discrete-event simulation messages

The procedures shown previously demonstrate the interaction of the messages in the object-oriented simulation software with the objects. This section provides details of the specific messages used for discrete-event simulation. Figure 3-32 lists all the messages used in the simulation software.

The procedures shown in Figures 3-33 through 3-36 show the specific messages for each class. Note how the messages are transformed into specific

```
procedure SimClass(MsgPacket:MsgPacketType);
{ interface to the outside world }
begin
    MData:=MsgPacket;
    case MData.Message of
    CLEAR_OBJ:        ClearAllObjects;      { clear all data from objects }
    DELETE_OBJ:       if DeleteCurrObject then ShowObject; { delete object instance }
    ENTER_OBJ:        GetObject(2);         { enter object instance data }
    LOAD_OBJ:         LoadObjects;          { load simulation objects from disk }
    SAVE_OBJ:         SaveObjects;          { save simulation objects to disk }
    SHOW_CURR_OBJ:    ShowObject;           { show current object instance }
    SHOW_NEXT_OBJ:    if GetNextObject then ShowObject; { show next object instance }
    SHOW_PREV_OBJ:    if GetPrevObject then ShowObject; { show prev object instance }
    UPDATE_OBJ:       GetObject(1);         { update object instance data }
    UPDATE_CLOCK:     UpdateClock;          { update simulation clock }
    REPORT_SIM:       ReportSimulation;     { print object detail }
    ENTITY_DEP:       EntityDeparted;       { entity has left system }
    end;
end;
```

Figure 3-33. Simulation class messages

```
procedure EntClass(MsgPacket:MsgPacketType);
{ interface to the outside world }
begin
    MData:=MsgPacket;
    case MData.Message of
    CLEAR_OBJ:        ClearAllObjects;      { clear all data from objects }
    DELETE_OBJ:       if DeleteCurrObject then ShowObject; { delete object instance }
    LOAD_OBJ:         LoadObjects;          { load simulation objects from disk }
    SAVE_OBJ:         SaveObjects;          { save simulation objects to disk }
    SHOW_CURR_OBJ:    ShowObject;           { show current object instance }
    SHOW_NEXT_OBJ:    if GetNextObject then ShowObject; { show next object instance }
    SHOW_PREV_OBJ:    if GetPrevObject then ShowObject; { show prev object instance }
    GEN_ARRIVAL:      GenerateArrival;      { generate an entity arrival }
    REQ_SQ_GRANTED:   RequestServQueGranted; { request for service/queue granted }
    REQ_SQ_DENIED:    RequestServQueDenied; { request for service/queue denied }
    ENTITY_SQ_COMP:   ServQueComplete; { service/queue completed, need next route }
    ENTITY_SET_FAIL:  SetFail(True);        { set entity to fail service }
    ENTITY_NO_FAIL:   SetFail(False);       { set entity to not fail service }
    LEAVE_SYS:        LeaveSystem;          { entity leaves simulation }
    end;
end;
```

Figure 3-34. Entity class messages

```
procedure RteClass(MsgPacket:MsgPacketType);
{ interface to the outside world }
begin
    MData:=MsgPacket;
    case MData.Message of
    CLEAR_OBJ:  ClearAllObjects;      { clear all data from objects }
    DELETE_OBJ: if DeleteCurrObject then ShowObject; { delete object instance }
    ENTER_OBJ:  GetObject(2);         { enter object instance data }
    LOAD_OBJ:   LoadObjects;          { load simulation objects from disk }
    SAVE_OBJ:   SaveObjects;          { save simulation objects to disk }
    SHOW_CURR_OBJ:   ShowObject;      { show current object instance }
    SHOW_NEXT_OBJ:   if GetNextObject then ShowObject; { show next object instance }
    SHOW_PREV_OBJ:   if GetPrevObject then ShowObject; { show prev object instance }
    UPDATE_OBJ:   GetObject(1);       { update object instance data }
    GEN_ARR_TIME: GenerateArrivalTime; { determine which arrival to generate & when }
    GET_NEXT_RTE: GetNextRoute(0);    { get next routing for an entity }
    GET_ALT_RTE:  GetNextRoute(1);    { get next routing for an entity, denied before }
    GET_FAIL_RTE: GetNextRoute(2);    { get failure route for an entity, failed service }
    GET_FAIL_RTRY: GetNextRoute(3);   { get failure route retry }
    SCH_SQ_COMP:  ScheduleSrvQueCompletion; { schedule service/queue completion }
    end;
end;
```

Figure 3-35. Routing class messages

```
procedure SrvClass(MsgPacket:MsgPacketType);
{ interface to the outside world }
begin
    MData:=MsgPacket;
    case MData.Message of
    CLEAR_OBJ:      ClearAllObjects; { clear all data from objects }
    DELETE_OBJ:     if DeleteCurrObject then ShowObject; { delete object instance }
    ENTER_OBJ:      GetObject(2);    { enter object instance data }
    LOAD_OBJ:       LoadObjects;     { load simulation objects from disk }
    SAVE_OBJ:       SaveObjects;     { save simulation objects to disk }
    SHOW_CURR_OBJ:  ShowObject;      { show current object instance }
    SHOW_NEXT_OBJ:  if GetNextObject then ShowObject; { show next object instance }
    SHOW_PREV_OBJ:  if GetPrevObject then ShowObject; { show prev object instance }
    UPDATE_OBJ:     GetObject(1);                { update object instance data }
    REPORT_SIM:     ReportSimulation;            { print object detail }
    REQ_SQ_ENTRY:   RequestServiceQueueEntry;    { entity is requesting entry }
    SQ_COMPLETE:    SrvQueCompletion;            { service/queue completion }
    ENTITY_LEAVE_SQ: EntityLeaveSrvQue;          { tell service/queue that entity has left }
    end;
end;
```

Figure 3-36. Server/queue class messages

procedures through the use of the Pascal case statement. Using the case statement with messages in this fashion allows the use of standard Pascal in conjunction with object-oriented programming techniques. The inherent speed and flexibility of the structured language is thus maintained. Note that only 30 messages are required for the entire simulation system.

G. Simulation Message Flow

The proper function of the object-oriented simulation program relies heavily on the correct sequence of message passing between objects. The previous section showed the specific messages used in the simulation program. This section demonstrates the flow of messages during execution of the simulation program.

1. Generating arrivals

The first activity that must take place when the simulation is started is the generation of the first arrival. The simulation clock is initialized to time zero. The main loop of the program then issues a message to the routing object to schedule the arrival of the next entity. Figure 3-37 shows the method that is invoked when the GEN_ARR_TIME message is sent to the routing object.

Note how the method shown in Figure 3-37 generates an arrival time for the future arrival of an entity. As stated previously, messages are placed in a single queue ordered by the clock time of the message. Messages will not be passed on until the current simulation clock is equal to or greater than the message time. Arrival times are generated for the future and the associated message to actually generate the arrival, GEN_ARRIVAL, is placed in the message queue to be executed at some future time. Scheduling of arrivals in this fashion creates an

```
procedure GenerateArrivalTime;
{ determine which arrivals to generate & when (entity types if Number = 0.0) }
var
    ATime:      real;
    Found:      boolean;
begin
    { find the first route record for the desired entity instance & type }
    TPtr: = PointTo(NINST,MData.Number);
    while TPtr < > nil do begin
        CurrObj: = TPtr;      { display the object for reference }
        PutObjInBuffer;
        ShowObject;
        { generate the arrival along with time (offset by simulation clock) }
        ATime: = GetDistNumber(TPtr ^ .Dist,TPtr ^ .Mean,TPtr ^ .Std);
        { send message indicating that an entity of should be generated }
        SendMsg(ROUTING,NINST,ENTITY,NINST,
            GEN_ARRIVAL,TPtr ^ .EntType,MData.Clock + ATime);
        { generate additional arrivals if desired }
        if MData.Number < 0.0 then Exit;
        repeat
            TPtr: = TPtr ^ .Next;
            if TPtr < > nil then Found: = (TPtr ^ .CurrLoc = NINST);
        until ((TPtr = nil) or (Found));
    end;
end;
```

Figure 3-37. GEN_ARR_TIME method in routing class

ordered queue of arrivals in a simulated future events calendar. Synchronization is automatically maintained.

Figure 3-38 shows the method invoked when the GEN_ARRIVAL message is sent from the routing objects to the entity objects. Entity objects are created dynamically when the GEN_ARRIVAL message is invoked. The creation time and other pertinent data are recorded for the entity. The entity object then issues a message GET_NEXT_ROUTE to the routing objects to determine where the entity should go. The entity object also issues another GEN_ARRIVAL message at the end of this method to schedule the next arrival, thus keeping the system moving.

```
procedure GenerateArrival;
{ generate an arrival of an entity }
begin
    { create a new entity }
    if not GetNewObject then Exit;
    { mark it's Instance id, arrival time, type code, status, etc... }
    CurrObj^.TypeCode:=MData.Number;
    CurrObj^.CreateTime:=MData.Clock;
    PutObjInBuffer;
    ShowObject;
    { request routing for self: "Where do I go?" }
    SendMsg(ENTITY,TPtr^.Instance,ROUTING,TPtr^.CurrLoc,
        GET_NEXT_RTE,TPTr^.TypeCode,SimClock);
    { generate next arrival of self }
    SendMsg(ENTITY,NINST,ROUTING,NINST,
        GEN_ARR_TIME,TPtr^.TypeCode,SimClock);
end;
```

Figure 3-38. GEN_ARRIVAL method in entity class

2. Routing entities

After an entity object has been generated it must be routed to a queue or a service. All information regarding the route of an entity through the system is contained in the routing class. The ordered nature of the message queue guarantees that there will be a message to the routing objects requesting the next route for an entity after the entity has been created. This message is GET_NEXT_RTE. Figure 3-39 shows the method invoked in the route class upon receipt of the GET_NEXT_RTE message.

The GET_NEXT_RTE method is the most complicated of all the messages in the simulation system.There are four codes that can be passed to this method depending on the previous state of the entity requesting a route. Initially, an entity request a primary route. If the primary route is blocked when the request for entry to a service or queue is made, then a balk route will be requested. In addition, entities may be predestined to fail service. Under failure conditions, a failure route may be specified by the user and that route will be requested when the failure occurs.

```
procedure GetNextRoute(RouteCode:byte);
{ get the next routing for an entity and send appropriate messages }
{       RouteCode:      0 - Get primary next location }
{                       1 - Get alternate route after denial of primary }
{                       2 - Get fail route after failure of service }
{                       3 - Retry getting fail route after denial }
begin
    { find the first route record for the desired entity instance & type }
    TPtr:=PointTo(MData.ToInst,MData.Number);
    if TPtr=nil then Exit;
    CurrObj:=TPtr;
    PutObjInBuffer;
    ShowObject;
    { if next location is blank, then leave system, otherwise request entry to location }
    if (((TPtr^.NextLoc=NINST) and (RouteCode in [0,1])) or
        ((TPtr^.FailTo=NINST) and (RouteCode in [2,3]))) then begin
        SendMsg(ROUTING,NINST,ENTITY,MData.FromInst,
            LEAVE_SYS,0.0,SimClock);
    end
    else case RouteCode of
        0:begin     { no prior denials, try first location }
            if TPtr^.BalkLoc=NINST then
                SendMsg(ENTITY,MData.FromInst,SERVQUE,TPtr^.NextLoc,
                    REQ_SQ_ENTRY,0.0,SimClock)
            else
                SendMsg(ENTITY,MData.FromInst,SERVQUE,TPtr^.NextLoc,
                    REQ_SQ_ENTRY,1.0,SimClock)
        end;
        1:if TPtr^.BalkLoc=NINST then begin
            { prior request failed, retry with clock incremented to next completion time }
            SendMsg(ENTITY,MData.FromInst,SERVQUE,TPtr^.NextLoc,
                REQ_SQ_ENTRY,0.0,SimClock);
        end
        else begin  { prior request denied, try alternate route }
            SendMsg(ENTITY,MData.FromInst,SERVQUE,TPtr^.BalkLoc,
                REQ_SQ_ENTRY,0.0,SimClock);
        end;
        2:begin     { service failed, request failure route }
            SendMsg(ENTITY,MData.FromInst,SERVQUE,TPtr^.FailTo,
                REQ_SQ_ENTRY,0.0,SimClock);
        end;
        3:begin     { service failed, request failure route repeated }
            SendMsg(ENTITY,MData.FromInst,SERVQUE,TPtr^.FailTo,
                REQ_SQ_ENTRY,0.0,SimClock);
        end;
    end;
end;
```

Figure 3-39. GET_NEXT_RTE method in routing class

Whenever an entity is to be routed to a service or queue, the REQ_SQ_ENTRY message is placed on the message queue at the current simulation clock time. The route code to use is determined in the server/queue class after the initial request for entry is made.

If there is no next route for an entity then the entity will be forced to leave the system when the routing objects issue the LEAVE_SYS message. The process followed when an entity leaves the system is described later.

3. Requesting service or queue entry

The routing objects issue the REQ_SQ_ENTRY message to the server/queue objects with data indicating the service or queue to request. The method invoked in the server/queue class when the REQ_SQ_ENTRY message is received is shown in Figure 3-40. The capacity of the requested service/queue is checked, and if space is available, the REQ_SQ_GRANTED message is sent to the entity object. If the request is denied, then the server/queue object determines if an alternate route is available or if the current request should be rescheduled. Note that if the current request is rescheduled, the request is delayed until the next completion time to avoid deadlocks in the system. If an alternate route should be tried, then the REQ_SQ_DENIED message is sent to the entity.

Figures 3-41 and 3-42 show the methods invoked by the entity object upon receipt of the REQ_SQ_GRANTED or REQ_SQ_DENIED messages. If entry to the service or queue is granted the entity first sends a message to the previous location that the entity is leaving. Next, a message is sent to the routing class to request the end of service time for the new location. Figure 3-43 shows the method invoked when the entity issues the ENTITY_LEAVE_SQ to the prior service or queue object. Note that this method issues no further messages.

```
procedure RequestServiceQueueEntry;
{ an entity is requesting entry }
begin
    TPtr:=PointTo(MData.ToInst);
    if TPtr=nil then Exit;
    if TPtr^.CurrQty<TPtr^..Capacity then begin     { entry is granted }
        { set to busy status }
        TPtr^.SrvStatus:=BUSY;
        TPtr^.CurrQty:=TPtr^.CurrQty+1.0;  { increase current contents by one }
        TPtr^.TotalQty:=TPtr^.TotalQty+1.0; { increase total quantity by one }
        { check for max quantity }
        if TPtr^.CurrQtyTPtr^.MaxQty then TPtr^.MaxQty:=TPtr^.CurrQty;
        { check for min interarrival time }
        if   (((SimClock-TPtr^.LastArrival)>0.0) and
             (((SimClock-TPtr^.LastArrival)>.MinTBA) or
             (TPtr^.MinTBA=0.0)))
             then TPtr^.MinTBA:=(SimClock-TPtr^.LastArrival);
        { check for max interarrival time }
        if (SimClock-TPtr^.LastArrival)>TPtr^.MaxTBA then
            TPtr^.MaxTBA:=(SimClock-TPtr^.LastArrival);
        { set mean time between arrivals }
        TPtr^.MeanTBA:=((TPtr^.MeanTBA*(TPtr^.TotalQty-1.0))+
                (SimClock-TPtr^.LastArrival))/TPtr^.TotalQty;
        { set last arrival time }
        TPtr^.LastArrival:=SimClock;
        { send message indicating request was granted }
        SendMsg(SERVQUE,TPtr^.Instance,ENTITY,MData.FromInst,
            REQ_SQ_GRANTED,0.0,SimClock);
    end
    else begin    { send message indicating request denied }
        if MData.Number=0.0 then   { no alternate, retry current }
            SendMsg(SERVQUE,TPtr^.Instance,ENTITY,MData.FromInst,
                REQ_SQ_DENIED,0.0,NextCompTime(TPtr^.Instance))
        else      { there is an alternate route }
            SendMsg(SERVQUE,TPtr^.Instance,ENTITY,MData.FromInst,
                REQ_SQ_DENIED,0.0,SimClock);
    end;
    CurrObj:=TPtr;
    PutObjInBuffer;
    ShowObject;
end;
```

Figure 3-40. REQ_SQ_ENTRY method in server/queue class

```
procedure RequestServQueGranted;
{ request for service/queue was granted, move entity to new location }
begin
    TPtr:=PointTo(MData.ToInst);
    if TPtr=nil then Exit;
    { send message to prior location that entity is leaving }
    SendMsg(ENTITY,TPtr^.Instance,SERVQUE,TPtr^.CurrLoc,
        ENTITY_LEAVE_SQ,SimClock-TPtr^.StartTime,SimClock);
    { set service failure flag off }
    TPtr^.WillFail:=False;
    { set current location }
    TPtr^.CurrLoc:=MData.FromInst;
    { set current location start time }
    TPtr^.StartTime:=SimClock;
    CurrObj:=TPtr;
    PutObjInBuffer;
    ShowObject;
    { send return message to indicate that entity is moved and completion should be scheduled }
    SendMsg(ENTITY,TPtr^.Instance,ROUTING,TPtr^.CurrLoc,
        SCH_SQ_COMP,TPtr^.TypeCode,SimClock);
end;
```

Figure 3-41. REQ_SQ_GRANTED method in entity class

```
procedure RequestServQueDenied;
{ request for service/queue was denied, attempt to reschedule/reroute }
begin
    TPtr:=PointTo(MData.ToInst);
    if TPtr=nil then Exit;
    CurrObj:=TPtr;
    PutObjInBuffer;
    ShowObject;
    if TPtr^.WillFail then begin        { entity was destined to fail service }
    SendMsg(ENTITY,TPtr^.Instance,ROUTING,TPtr^.CurrLoc,
        GET_FAIL_RTRY,TPTr^.TypeCode,SimClock);
    end
    else begin    { request alternate routing for entity }
    SendMsg(ENTITY,TPtr^.Instance,ROUTING,TPtr^.CurrLoc,
        GET_ALT_RTE,TPTr^.TypeCode,SimClock);
    end;
end;
```

Figure 3-42. REQ_SQ_DENIED method in entity class

```
procedure EntityLeaveSrvQue;
{ tell service/queue that entity is leaving }
begin
    TPtr:=PointTo(MData.ToInst);
    if TPtr=nil then Exit;
    { set avg contents and utilization }
    TPtr^.AvgQty:=(((TPtr^.TotalQty-1.0)*TPtr^.AvgQty)+
        TPtr^.CurrQty)/TPtr^.TotalQty;
    TPtr^.Utilized:=TPtr^.AvgQty*100.0/TPtr^.Capacity;
    { reduce current quantity by one }
    TPtr^.CurrQty:=TPtr^.CurrQty-1.0;
    if TPtr^.CurrQty then TPtr^.SrvStatus:=IDLE;
    { set min time here }
    if ((MData.Number>0.0) and ((MData.Number<TPtr^.MinTime) or
        (TPtr^.MinTime=0.0))) then TPtr^.MinTime:=MData.Number;
    { set max time here }
    if MData.NumberTPtr^.MaxTime then TPtr^.MaxTime:=MData.Number;
    { set mean time here }
    TPtr^.MeanTime:=((TPtr^.MeanTime*TPtr^.TotalQty)+
        MData.Number)/(TPtr^.TotalQty+1.0);
    CurrObj:=TPtr; PutObjInBuffer; ShowObject;
end;
```

Figure 3-43. ENTITY_LEAVE_SQ method in server/queue class

4. Scheduling completions

Figure 3-44 shows the method invoked when the routing object receives the SCH_SQ_COMP message. Data are contained in the routing object for the current entity that indicate the distribution to use to determine the end of service time for an entity in a particular service. Entities in queues have no set completion time and will proceed to the next routed location as soon as possible.

The software created in this research supports Uniform, Normal, and Exponential distributions for service time. After the routing object has determined the completion time for an entity, a SQ_COMPLETE message is sent to the server/queue object with the clock field set to the completion time. This message will not be invoked until the simulation clock reaches the designated time. The result is that the entity will remain in the service or queue until the desired simulation time is reached. Failure percentages are examined in the

```
procedure ScheduleSrvQueCompletion;
{ schedule service/queue completion }
var
    ATime:      real;
begin
    { find the route record for the desired entity instance & type }
    TPtr:=PointTo(MData.ToInst,MData.Number);
    if TPtr=nil then Exit;
    CurrObj:=TPtr;
    PutObjInBuffer;
    ShowObject;
    { generate the completion time (offset by simulation clock) }
    ATime:=GetDistNumber(TPtr^.Dist,TPtr^.Mean,TPtr^.Std)+SimClock;
    if GetDistNumber('UNFRM',50.0,50.0) < TPtr^.FailPerc then
        { set entity for service failure }
        SendMsg(ROUTING,NINST,ENTITY,MData.FromInst,
            ENTITY_SET_FAIL,0.0,ATime);
    { send message to schedule service/queue completion }
    SendMsg(ROUTING,MData.FromInst,SERVQUE,TPtr^.CurrLoc,
        SQ_COMPLETE,0.0,ATime);
end;
```

Figure 3-44. SCH_SQ_COMP method in routing class

```
procedure SetFail(Fail:boolean);
{ set entity fail service flag }
begin
    TPtr:=PointTo(MData.ToInst);
    if TPtr=nil then Exit;
    { update entity statistics here }
    TPtr^.WillFail:=Fail;
    CurrObj:=TPtr;
    PutObjInBuffer;
    ShowObject;
end;
```

Figure 3-45. ENTITY_SET_FAIL method in entity class

SCH_SQ_COMP method. If the entity is to fail the ENTITY_SET_FAIL message is sent to the entity to set the failure flag. This information is used in the routing request methods described earlier. Figure 3-45 shows the method invoked in the entity class upon receipt of the ENTITY_SET_FAIL message.

```
procedure SrvQueCompletion;
{ service/queue completion }
begin
    TPtr: = PointTo(MData.ToInst);
    if TPtr = nil then Exit;
    CurrObj: = TPtr;
    PutObjInBuffer;
    ShowObject;
    { send message to indicate completion and next route is required }
    SendMsg(SERVQUE,TPtr ^ .Instance,ENTITY,MData.FromInst,
        ENTITY_SQ_COMP,0.0,SimClock);
end;
```

Figure 3-46. SQ_COMPLETE method in server/queue class

```
procedure ServQueComplete;
{ service/queue completed, need next route }
begin
    TPtr: = PointTo(MData.ToInst);
    if TPtr = nil then Exit;
    CurrObj: = TPtr;
    PutObjInBuffer;
    ShowObject;
    if TPtr ^ .WillFail then begin        { entity was destined to fail service }
        SendMsg(ENTITY,TPtr ^ .Instance,ROUTING,TPtr ^ .CurrLoc,
            GET_FAIL_RTE,TPTr ^ .TypeCode,SimClock);
    end
    else begin        { send return message requesting next route }
    SendMsg(ENTITY,TPtr ^ .Instance,ROUTING,TPtr ^ .CurrLoc,
        GET_NEXT_RTE,TPtr ^ .TypeCode,SimClock);
    end;
end;
```

Figure 3-47. ENTITY_SQ_COMP method in entity class

5. Completing service

Figure 3-46 shows the method invoked when the server/queue object receives the SQ_COMPLETE message. Data contained in the server/queue object are modified and a message is sent to the entity object indicating completion of service. Figure 3-47 shows the method invoked when the entity object receives the ENTITY_SQ_COMP message.

When the ENTITY_SQ_COMP message is sent to the entity object, the entity initiates a request for the next routing location with either the GET_NEXT_RTE message or the GET_FAIL_RTE message depending on the status of the failure flag. With the invocation of the request for next routing location, the simulation has completed a cycle.

6. Leaving the system

If the next route for an entity is blank in the routing object, the entity will be instructed to leave the system with the LEAVE_SYS message from the routing object. Figure 3-48 shows the method invoked when the entity object receives the LEAVE_SYS message. The entity object again uses the ENTITY_LEAVE_SQ message described earlier to inform the server/queue object that an entity is leaving the current location. In addition, the entity object issues the ENTITY_DEP message to the simulation object to force simulation statistics collection. Figure 3-49 shows the method invoked when the simulation object receives the ENTITY_DEP message. Note that there are no messages generated by the ENTITY_DEP method in the simulation class.

This chapter presented the structure of the simulation program. Through the use of object-oriented programming the simulation system was constructed with only 4000 lines of Pascal code. The entire system, capable of a wide range of discrete-event simulations, operates with only 30 different messages passed between objects.

The next step in the use of the simulation program is the actual execution of the software. The next chapter provides operational details of the simulation software.

```
procedure LeaveSystem;
{ entity leaves simulation }
begin
    TPtr:=PointTo(MData.ToInst);
    if TPtr=nil then Exit;
    { send message to prior location that entity is leaving }
    SendMsg(ENTITY,TPtr^.Instance,SERVQUE,TPtr^.CurrLoc,
        ENTITY_LEAVE_SQ,SimClock-TPtr^.StartTime,SimClock);
    { update entity statistics here }
    TPtr^.CurrLoc:=NINST;
    TPtr^.TimeInSys:=SimClock-TPtr^.CreateTime;
    CurrObj:=TPtr;
    PutObjInBuffer;
    ShowObject;
    { send message indicating entity throughput }
    SendMsg(ENTITY,TPtr^.Instance,SIMULATE,NINST,
        ENTITY_DEP,TPtr^.TimeInSys,SimClock);
    { delete the entity, no longer needed }
    if DeleteCurrObject then;
end;
```

Figure 3-48. LEAVE_SYS method in entity class

```
procedure EntityDeparted;
{ an entity has left the system }
begin
    { set throughput }
    CurrObj^.CurrQty:=CurrObj^.CurrQty+1.0;
    { set min time in system }
    if ((MData.Number>0.0) and ((MData.Number>CurrObj^.MinTInSys) or
        (CurrObj^.MinTInSys=0.0))) then
            CurrObj^.MinTInSys:=MData.Number;
    { set max time in system }
    if MData.NumberCurrObj^.MaxTInSys then CurrObj^.MaxTInSys:=MData.Number;
    { set avg time in system }
    CurrObj^.AvgTInSys:=((CurrObj^.AvgTInSys*(CurrObj^.CurrQty-1.0)) +
        MData.Number)/CurrObj^.CurrQty;
    PutObjInBuffer;
    ShowObject;
end;
```

Figure 3-49. ENTITY_DEP method in simulation class

IV. SIMULATION PROGRAM OPERATION

A. Introduction

The previous chapter described the internal operation of the object-oriented simulation program developed in this research. This chapter provides information to assist the user in the operation of the actual program.

Requirements of the program and procedures to start the program are outlined first. Program menus are then described, followed by a presentation of data entry methods. Commands used to load and save simulation data are given. The balance of the chapter is devoted to techniques of running simulations using the simulation program.

B. Starting the Program

The program is written in Turbo Pascal version 5.0 and is designed to operate on standard MS-DOS microcomputers. The program requires the host machine to have at least 320K of random access memory (RAM). In addition, the host machine must have at least one floppy disk drive. A fixed disk is recommended for optimal program operation. The program has been compiled to a standard executable ".EXE" file and does not require any additional interpreters to operate.

The name of the simulation program file is "SOOP.EXE." Several support files with filename extensions of ".DBD", ".C01", ".C02", ".C03", and ".C04" are located on the distribution disk. The "SOOP.EXE" file and all support files must be present for correct operation of the program. The user should transfer all files found on the distribution disk to a subdirectory on a hard disk or to a separate floppy disk before attempting to execute the program.

After the correct files have been transferred to the desired location the user may start the simulation program by entering the "SOOP" command from the DOS command line. A brief delay will occur while the program is loaded into computer memory and all initialization is completed. The main program screen will then appear.

The top line of the screen shows the name of the program, the current message queue count, and the name of the simulation currently in memory. The current simulation name will initially be blank. The center portion of the screen displays one of the four class screens. The classes are "SIMULATION CLASS", "ENTITY CLASS", "ROUTING CLASS", and "SERVER/QUEUE CLASS." Each of these classes contains different information which is displayed to the user. Only one class type is displayed at one time.

The user may change the currently displayed class by pressing the PgUp or PgDn keys. Each keypress moves to the next or previous class. The display of object classes is a circular linked list so repeated PgUp or PgDn keystrokes will rotate through the classes.

Only one instance of an object will appear on the screen at one time. Initially, there will be no instances of any of the objects. After the user loads a simulation from disk or enters data manually, there will be some instances of some of the objects. The user may view different instances of an object by pressing the F5 or F6 function keys. The F5 key will display the previous instance of an object and the F6 key will display the next instance of an object. The order of instances of objects depends on their order of creation. Using the keystrokes described above, the user can quickly move from class to class and from object to object.

The bottom portion of the screen displays the program menus. The use of the program menus is described in the next section.

C. Program Menus

Most of the program functions are executed through the list of commands shown on the bottom portion of the screen. The "space", "backspace", "left arrow", or "right arrow" keys may be used to highlight the desired command. A single line of text describing the highlighted command will be shown on the last line of the screen. To execute any of the program commands, the user may either highlight the desired command and press the "enter" key or press the first letter of the desired command.

Some of the menus shown on the bottom of the screen will not allow movement of a highlight bar. These command lists are distinguished by the absence of a highlight bar on any one command. The user may select a command from this type of menu by pressing the indicated letter or function key.

The selection of some commands will display another command list. The user may move to a previous command list by pressing the "esc" key. Each command list also has a "quit" command which will also serve to move to the previous command list.

Occasionally the user will be presented with a vertical list of choices for some of the program options. Selections from these lists are made by pressing the "up arrow" or "down arrow" keys to highlight the desired option followed by pressing the "return" key.

To quit the program and return to the DOS prompt the "quit" command found on the main program menu is selected. Alternately, the "esc" key may be used to quit from the program.

D. Object Definition

To correctly configure the program for simulations the user must define the objects for the desired simulation. There are four classes in this simulation program. The user must enter data in the "simulation", "routing", and "server/queue" classes. The "entity" class does not allow the user to enter data or create instances of an entity object.

1. Data entry basics

The "Enter" command is used to enter new object instances to the program. The "Update" command is used to update existing object instances. The user should make sure the desired object is shown on the screen before using the "Enter" or "Update" commands.

When entering data or updating data screens a flashing "*" will appear on the data screen to indicate where information is to be entered. A cursor will also show the current position within a data field. Data fields are the highlighted areas in the center portion of the screen. Only some of the data fields may be edited by the user. The program automatically restricts data entry to allowable fields. The following commands are available for data entry:

- [left or right arrow]: Moves the cursor within the current field. If the cursor is at the first position within a field then the previous field will be selected. If the cursor is at the last position within a field then the next field will be selected.
- [ctrl-left or ctrl-right]: Moves the cursor to the first position or to the last character of a field.
- [up or down arrow]: Selects the previous or next data field for editing.
- [Enter]: Same as right arrow for selecting the next field.
- [F5]: Accepts changes made to the data screen and displays the previous object instance. This command is only available in update mode.

- [F6]: Accepts changes made to the data screen and displays the next object instance. If in enter mode, the new data screen will be void of data.
- [F8]: Blanks the current data field.
- [Esc]: Aborts changes made to the data screen and completes the enter/update action.
- [F10]: Accepts changes made to the data screen and completes the enter/update action.

Data are required in all but the entity class of objects. The following sections describe the data screens for each of the object classes.

2. Simulation class

Simulation class data are used to restrict the total run time for a simulation and to track general simulation statistics. Only one instance of a simulation class object is used to define a simulation for the program. Additional simulation object instances that are entered will be ignored by the program. The following data fields are available in simulation objects.

- Simulation Instance: This field contains a short name for the simulation instance.
- Description: This field is used to describe the simulation instance.
- Maximum Time: This field contains the maximum time that the simulation is allowed to run measured in simulation time, not real time. When the simulation clock reaches this time the simulation will automatically stop.
- Current Time: This field shows the current value of the simulation clock. The user may not modify this field.
- Current Throughput: This field shows the total number of entities that have been processed through the system. This number includes all entities that have left the system regardless of the exit route. The user may not modify this field.

- Time In System (Min, Max, Avg): This field shows the minimum, maximum, and average times that entities spend in the system. The user may not modify these fields.

3. Entity class

The entity class contains instances of the entities that flow through the simulation. Entities are automatically created and deleted by the software as required by the simulation and as specified by the user. The user may not manually create entities, but the user may view any currently existing entities. The following data fields are found in the entity class.

- Entity Instance: This field contains a short name for the entity instance.
- Entity Type Code: This field contains a number used to identify the general type of entity. The software allows multiple types of entities to be simulated. The entity types are classified in the routing class described later.
- Current Location: This field indicates in which server/queue object instance the entity currently resides.
- Will Service Fail?: This field is used to predetermine entity failures in a server/queue object instance. Failures are set by percentages in the routing class described later.
- Entity Creation Time: This field marks the simulation time at which the entity was created.
- Time Started Curr Loc: This field marks the simulation time at which the entity entered the current server/queue object instance.
- Total Time In System: This field indicates the total time the entity spent in the simulated system.

4. Server/queue class

The server/queue class contains an entry for each server and queue in the defined simulation. Any point in a simulation where statistics are desired or where

an entity is to spend some time must be defined as a server or queue object. The user is responsible for the correct definition of servers and queues for the desired simulation. The examples shown in the next chapter may serve as guidelines for the definition of servers and queues. The following data fields are available in the server/queue class.

- Server/Queue Instance: This field contains a short name for the server/queue object instance.
- Description: This field is used to describe the server/queue instance.
- Server/Queue Cap: This field is used to limit the total capacity for the server/queue instance. This field is normally set to 1 for servers but may be set higher if more than one identical server is available. This field should be set to a very large number if there is no limit on capacity.
- Current Quantity: This field shows the current number of entities in the server/queue. The user may not modify this field.
- Maximum Quantity: This field shows the largest number of entities contained in the server/queue during the current simulation. The user may not modify this field.
- Average Quantity: This field shows the average number of entities contained in the server/queue during the current simulation. The user may not modify this field.
- Total Throughput: This field shows the total number of entities processed through the server/queue during the current simulation. The user may not modify this field.
- Percent Utilization: This field shows the utilization of the server/queue during the current simulation. The user may not modify this field.
- Status: This field shows the current status of the server/queue. If the current quantity in the server/queue is less than the capacity, the status will show the "idle" indicator. If the current quantity in the server/queue is at capacity, the status will show the "busy" indicator. The user may not modify this field.
- TBA (Minimum, Maximum, Mean): These fields show the minimum, maximum, and mean time between arrivals at the server/queue. The user may not modify these fields.

- Time Spent Here (Minimum, Maximum, Mean): These fields show the minimum, maximum, and mean time that entities have spent in this server/queue. The user may not modify these fields.
- Last Time Arrival Occurred: This field shows the last simulation time that an entity arrived at this server/queue. The user may not modify this field.

5. Routing class

After the user has defined the servers and queues contained in the desired simulation, the relationships between servers and queues must be defined. The routing objects are used to define the paths that entities will take through the simulated system. Routing objects are also used to define the amount of time (process time) that entities will spend at each server object. The user may modify any of the fields in the routing class. The following data fields are available in the routing class.

- Routing Instance: This field contains a short name for the routing instance.
- Desc: This field is used to describe the routing instance.
- Ent Type: This field is used to designate the type of entity to which this routing instance applies.
- Current Location: This field contains the server/queue instance name which is described by the routing object instance. For initial creation of entities, this field should be left blank.
- Stay At Current Location (Distribution, Mean, Range or Std Dev): These fields are used to describe the amount of time that an entity should remain in the server/queue named in the "Current Location" field. The distribution may be "UNFRM" for the uniform distribution, "EXPON" for the exponential distribution, or "NORML" for the normal distribution. The mean and range must be specified for the uniform distribution. The mean must be specified for the exponential distribution. The mean and standard deviation must be specified for the normal distribution.

- Failure Percent: This field is used to specify the percentage of entities that will fail the service named in the "Current Location" field. This field is useful to create a routing split between two alternate paths. Several routing instances with failure percentages may be linked together to provide multiple split options.
- Failures Go To: This field is used to designate the next server/queue for an entity that fails the current server/queue.
- Successes Go To: This field is used to designate the next server/queue for an entity that succeeds the current server/queue. If the entity is to leave the system after the current server/queue, this field should be left blank.
- Balks Go To: This field is used to designate the next location for an entity that is not allowed to enter the "Successes Go To" server/queue because the next location is at capacity. If balking is not allowed, this field should be left blank. Blocked entities would then retry the server/queue specified in the "Successes Go To" field until entry is allowed.

Proper definition of the objects is essential to the correct operation of the simulation. If the user finds that the results of a simulation do not appear correct, the data in the object instances should be examined.

E. Loading and Saving the Simulation

The objects in the simulation program contain a large amount of data. Complex simulations may be comprised of many object instances. Specification of a simulation is time-consuming and if possible should not be repeated.

The software created in this research allows the user to save simulation specifications to a disk file to avoid repeating the data entry task. Simulations saved to disk may later be loaded to rerun the simulation. Completed simulations may be saved to disk to retain final or intermediate results for later review.

To save a simulation to disk, the user should select the "Save" command from the program command list. The user will be prompted for a file name to save the

simulation. The file name may be up to 8 characters. After the file name is entered the simulation will be saved to disk. If a simulation with the same name already exists on disk, the user will be asked if the existing simulation should be replaced.

To load an existing simulation from disk the user should select the "Load" command from the program command list. The user will be asked if the current simulation should be cleared. The user will then be prompted for the name of the simulation to load from disk. If the file name specified by the user exists, the simulation will be loaded into memory and will be prepared for execution. If the file name specified by the user does not exist, a new simulation will be created in memory.

F. Running the Simulation

After the user has specified the desired simulation or loaded an existing simulation specification from disk the simulation program will be prepared to execute the simulation. The following guidelines provide information to assist the user when running a simulation.

1. Starting the simulation

The "Proceed" command is used to start the simulation. The user should ensure that the simulation has been completely defined before starting the simulation. After the simulation has been started, the current message count indicator will show the number of messages in the message queue as the simulation proceeds. The command list will also change to show a "Pause" command instead of the "Proceed" command.

The object instance that is shown on the screen when the simulation is started will remain visible to the user during the simulation if single-step execution is not

enabled. The data contained in the current object instance will change as the simulation progresses. These data are shown to the user as they change.

2. Interrupting the simulation

At many times during the execution of a simulation the user may wish to stop the simulation to examine or modify object instances. The "Pause" command is used to interrupt the execution of a simulation. When the "Pause" command is selected, the simulation will stop and the command list will change to show the "Proceed" command instead of the "Pause" command.

Although other commands found on the command list may be used while a simulation is in progress, only the "Pause" command should be used. If other commands are used during simulation execution, the data in object instances may be in an undefined state and will not be reliable.

3. Changing the simulation

Data in object instances may be changed while the simulation is interrupted. Care should be used when altering instance data after a simulation has been started. Modification is accomplished with the "Update" command. New instances of an object may be added to the simulation with the "Enter" command. Changes to the structure of the simulation during execution is often enlightening when testing the effect of changes on the operation of a system.

4. Viewing alternate classes and objects

Under normal circumstances only a single class and object instance will be shown to the user during simulation execution. It is often desirable to examine other classes and object instances during a simulation. Selecting an alternate class or instance of an object can be performed in either the paused or active execution states.

The user may select an alternate class for display by pressing the "PgUp" or "PgDn" keys until the desired class is shown on the screen. Alternate instances of an object may be displayed by pressing the "F5" or "F6" function keys until the desired object instance is shown on the screen.

If alternate classes or object instances are selected while the simulation is in progress a slight delay may occur while messages with high priority are processed. Optimally, the simulation should be interrupted when changing to alternate classes or object instances.

5. Restarting the simulation

After the simulation has been interrupted with the "Pause" command it may be restarted with the "Proceed" command. Use of the "Proceed" command in this fashion will restart the simulation at the point it was interrupted.

If the user desires to restart the simulation from time zero, the "Clr" command should be used before the "Proceed" command. The "Clr" command clears all data from the object instances. An alternate method to restart a simulation from time zero is to use the "Load" command to load the same simulation from disk.

6. Single-step operation

Under the default simulation program parameters only the class and object instance displayed when the simulation is started will be shown to the user during program execution. To fully understand the object-oriented nature of the simulation program developed in this research it is helpful to see the messages and results of the messages as they are sent and received in the software. Viewing messages in this fashion is called "single-stepping".

Single-step execution of the simulation program is enabled or disabled with the "Options" command. When the "Options" command is selected from the

command list a vertical list of program options will appear. The user should use the "up arrow" or "down arrow to highlight the "Single-step" option and then press "return" to toggle single-step execution on or off.

After single-step execution has been enabled, each message sent from one object to another object will be shown at the bottom of the screen. When the message is received by the target object, the message will again be shown at the bottom of the screen. In addition, the target object instance will be shown to the user.

The use of single-step execution allows insight to the flow of messages in the object-oriented simulation software. Single-step execution drastically slows the execution of the program and should be avoided unless specifically desired.

7. Printing reports

Printed output from a simulation is obtained with the "Report" command. The simulation should be interrupted when the "Report" command is selected. The "Report" command sends a message to each of the object instances asking for a complete report of their contents. Each object will print a summary of its current instance variables when it receives the report request.

V. DATA COLLECTION AND ANALYSIS

A. Introduction

The major effort of this research is contained in the development of the object-oriented simulation program. However, a degree of research analysis is still required. There are two phases in the analysis portion of this research. First, the output produced by the proposed program is compared to the output produced by a traditional simulation language to verify the correctness of the object-oriented simulation program. Second, the operating characteristics of the simulation program will be compared to those of a traditional simulation language.

Three simulation models are presented for verification followed by a complicated simulation. Object-oriented simulation is then compared to traditional simulation.

B. Simulation Verification

Simulation software is complicated and difficult to create. The object-oriented simulation program developed in this research contains over 4000 lines of Pascal source code. While it is possible to examine the output of the simulation program for reasonableness, a more thorough method of program verification is to compare the output produced by this software with the output produced by established simulation language.

The simulation language used for comparison purposes is SLAM from Pritsker and Associates. SLAM is used because of its popularity, availability on microcomputers, and its reputation as a correctly functioning simulation program. SLAM's process-orientation allows direct comparison with the object-oriented simulation program developed in this research.

The object-oriented simulation program is not intended to be a complete simulation package but rather, to serve as a demonstration of the possibilities of object-oriented programming for simulation in a procedural language. The simulation program is capable of simulating systems with multiple servers and queues. Arrival and service distributions may be selected from the uniform, exponential, and normal family of distributions. Resource usage is not supported in the simulation program.

Four simulation models are used to test the object-oriented simulation program. The first three models are relatively simple and are designed to test the basic correctness of the program. The fourth model is a complex combination of simpler models and is used to demonstrate the advantages and capabilities of the object-oriented simulation program. Statistics collected from both simulation programs are presented for comparison.

1. Single-server model

Figure 5-1 shows the first simulation model used for verification. In this system entities arrive exponentially with a mean of 0.4 time units. The service is also exponentially distributed with a mean of 0.25 time units. Only four entities may wait in the queue. If the queue is full when an entity arrives, the entity will be sent to a subcontractor. The simulation is to be run for 300 time units. Figure 5-2 shows the data the user would enter in the objects of the object-oriented simulation program.

The SLAM model for the single-server system is shown in Figure 5-3. The system was simulated using both SLAM and the object-oriented simulation program. A comparison of results from the simulations is shown in Table 5-1. As shown, the results are similar under both simulation programs.

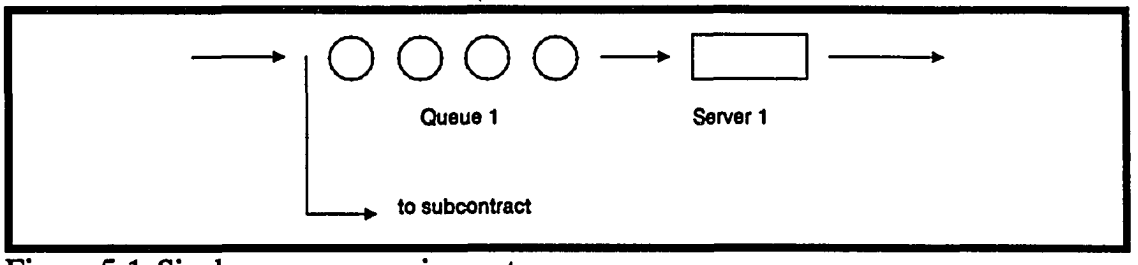

Figure 5-1. Single-server queueing system

Simulation: TEST1 **Description:** Single Server Queue **Maximum Time:** 300.00

Routing:	R1	**Desc:** Entity Creation	**Ent Type:** 1.0	**Current Location:**
Distribution:	EXPON	**Mean:** 0.40 **Range or Std Dev:** 0.00		**Failure Percent:** 0.00
Failures Go To:		**Successes Go To:** Q1		**Balks Go To:** SUBC
Routing:	R2	**Desc:** Enter Service 1	**Ent Type:** 1.0	**Current Location:** Q1
Distribution:		**Mean:** 0.00 **Range or Std Dev:** 0.00		**Failure Percent:** 0.00
Failures Go To:		**Successes Go To:** S1		**Balks Go To:**
Routing:	R3	**Desc:** Finish Service 1	**Ent Type:** 1.0	**Current Location:** S1
Distribution:	EXPON	**Mean:** 0.25 **Range or Std Dev:** 0.00		**Failure Percent:** 0.00
Failures Go To:		**Successes Go To:**		**Balks Go To:**
Routing:	R4	**Desc:** Finish Subcontract	**Ent Type:** 1.0	**Current Location:** SUBC
Distribution:		**Mean:** 0.00 **Range or Std Dev:** 0.00		**Failure Percent:** 0.00
Failures Go To:		**Successes Go To:**		**Balks Go To:**

Server/Queue:	Q1	**Description:** Queue Number 1	**Server/Queue Cap:** 4.00
Server/Queue:	S1	**Description:** Service Number 1	**Server/Queue Cap:** 1.00
Server/Queue:	SUBC	**Description:** Subcontracted Parts	**Server/Queue Cap:** 1.00

Figure 5-2. Object data for single-server model

```
GEN,DIESCH,SERIAL SINGLE SERVER,1/24/89,1;
LIMITS,2,2,50;
NETWORK;
        CREATE,EXPON(.4),,1;                    CREATE ARRIVALS
        QUEUE(1),0,4,BALK(SUB);                 STATION 1 QUEUE
        ACT/1,EXPON(.25);                       STATION 1 SERVER TIME
        COLCT,INT(1),TIME IN SYSTEM,20/0/.25;   COLLECT STATISTICS
        TERM;
SUB     COLCT,BET,TIME BET. BALKS;              COLLECT STATISTICS
        TERM;
        END
INIT,0,300;
FIN;
```

Figure 5-3. SLAM statements for single-server model

Table 5-1. Simulation results for single-server model

	SLAM	SOOP
Maximum length of queue 1	4.00	4.00
Average wait time in queue 1	0.34	0.36
Total throughput of server 1	700	710
Total units to subcontract	42	45
Mean time between balks	6.28	6.05
Mean time in system	0.60	0.57

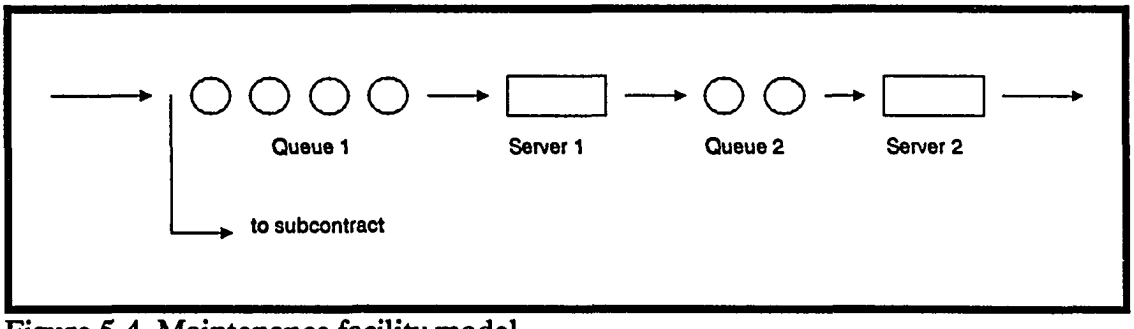

Figure 5-4. Maintenance facility model

2. Maintenance facility model

There are two operations performed on the entities in the maintenance facility shown in Figure 5-4. These operations are performed in series. Operation 2 always follows operation 1. The queue before work station 1 allows room for four units and there is space for two units in the queue preceding work station 2. If an arriving unit cannot enter the first queue it is sent to a subcontractor.

The interarrival time for units entering the maintenance facility is exponentially distributed with a mean of 0.4 time units. Service times are also exponentially distributed with a mean of 0.25 time units for work station 1 and a mean of 0.5 time units for work station 2. Transport time between work stations is

Simulation: TEST2 **Description:** Maintenance Facility **Maximum Time:** 300.00

Routing: R1 **Desc:** Entity Creation **Ent Type:** 1.0 **Current Location:**
Distribution: EXPON **Mean:** 0.40 **Range or Std Dev:** 0.00 **Failure Percent:** 0.00
Failures Go To: **Successes Go To:** Q1 **Balks Go To:** SUBC

Routing: R2 **Desc:** Enter Service 1 **Ent Type:** 1.0 **Current Location:** Q1
Distribution: **Mean:** 0.00 **Range or Std Dev:** 0.00 **Failure Percent:** 0.00
Failures Go To: **Successes Go To:** S1 **Balks Go To:**

Routing: R3 **Desc:** Finish Service 1 **Ent Type:** 1.0 **Current Location:** S1
Distribution: EXPON **Mean:** 0.25 **Range or Std Dev:** 0.00 **Failure Percent:** 0.00
Failures Go To: **Successes Go To:** Q2 **Balks Go To:**

Routing: R4 **Desc:** Enter Service 2 **Ent Type:** 1.0 **Current Location:** Q2
Distribution: **Mean:** 0.00 **Range or Std Dev:** 0.00 **Failure Percent:** 0.00
Failures Go To: **Successes Go To:** S2 **Balks Go To:**

Routing: R5 **Desc:** Finish Service 2 **Ent Type:** 1.0 **Current Location:** S2
Distribution: EXPON **Mean:** 0.50 **Range or Std Dev:** 0.00 **Failure Percent:** 0.00
Failures Go To: **Successes Go To:** **Balks Go To:**

Routing: R6 **Desc:** Finish Subcontract **Ent Type:** 1.0 **Current Location:** SUBC
Distribution: **Mean:** 0.00 **Range or Std Dev:** 0.00 **Failure Percent:** 0.00
Failures Go To: **Successes Go To:** **Balks Go To:**

Server/Queue:	Q1	**Description:** Queue Number 1	**Server/Queue Cap:** 4.00
Server/Queue:	S1	**Description:** Service Number 1	**Server/Queue Cap:** 1.00
Server/Queue:	Q2	**Description:** Queue Number 2	**Server/Queue Cap:** 2.00
Server/Queue:	S2	**Description:** Service Number 2	**Server/Queue Cap:** 1.00
Server/Queue:	SUBC	**Description:** Subcontracted Parts	**Server/Queue Cap:** 1.00

Figure 5-5. Object data for maintenance facility

```
GEN,DIESCH,MAINTENANCE FACILITY,1/24/89,1;
LIMITS,2,2,50;
NETWORK;
        CREATE,EXPON(.4),,1;                          CREATE ARRIVALS
        QUEUE(1),0,4,BALK(SUB);                       STATION 1 QUEUE
        ACT/1,EXPON(.25);                             STATION 1 SERVER TIME
        QUEUE(2),0,2,BLOCK;                           STATION 2 QUEUE
        ACT/2,EXPON(.50);                             STATION 2 SERVER TIME
        COLCT,INT(1),TIME IN SYSTEM,20/0/.25;         COLLECT STATISTICS
        TERM;
SUB     COLCT,BET,TIME BET. BALKS;                    COLLECT STATISTICS
        TERM;
        END
INIT,0,300;
FIN;
```

Figure 5-6. SLAM statements for maintenance facility

Table 5-2. Simulation results for maintenance facility

	SLAM	SOOP
Maximum length of queue 1	4.00	4.00
Average length of queue 1	1.99	2.60
Average wait time in queue 1	1.10	1.23
Maximum length of queue 2	2.00	2.00
Average length of queue 2	1.43	1.69
Average wait time in queue 2	0.79	0.76
Total throughput of server 1	541	540
Total throughput of server 2	538	537
Total units to subcontract	178	227
Mean time between balks	1.65	1.30
Mean time in system	2.87	2.06

negligible. If the queue for work station 2 is full, the first work station is blocked and units cannot leave that station. A blocked work station cannot serve other units until it is unblocked.

The maintenance facility is to be simulated for 300 time units. Figure 5-5 shows the data the user must enter in the objects. Figure 5-6 shows the SLAM statements required to simulate the maintenance facility. Table 5-2 shows the results of the simulation using both SLAM and the object-oriented simulation program. As shown, the results are similar.

3. TV inspection and adjustment model

In the system shown in Figure 5-7 assembled television sets move through a series of testing stations. A final test is performed at the last of these stations. If the sets fail the final test, the set is routed to an adjustment station where the set is adjusted. After adjustment, the television set is sent back to the last inspection

station where the set is again inspected. When the television set finally passes inspection it is routed to a packing area. There is no limit placed on the number of sets that may wait in any of the queues in the inspection system.

The time between arrivals of television sets for inspection is uniformly distributed between 3.5 and 7.5 minutes. There are two identical inspectors at the inspection station. The time required to inspect a set is uniformly distributed between 6 and 12 minutes. On the average, 85 percent of the set pass inspection and are routed to the packing area. The remaining 15 percent fail inspection and are sent to the adjustment station. Adjustment requires between 20 and 40 minutes, uniformly distributed.

The system is to be simulated for 480 minutes. Figure 5-8 shows the data the user must enter in the objects of the object-oriented simulation program. Figure 5-9 shows the SLAM statements required to simulate the model. Table 5-3 presents the results of the simulations using SLAM and the object-oriented simulation program. As shown, the results obtained from both programs are similar.

The three test simulations serve to verify the correct operation of the object-oriented simulation program. Under different conditions, the results

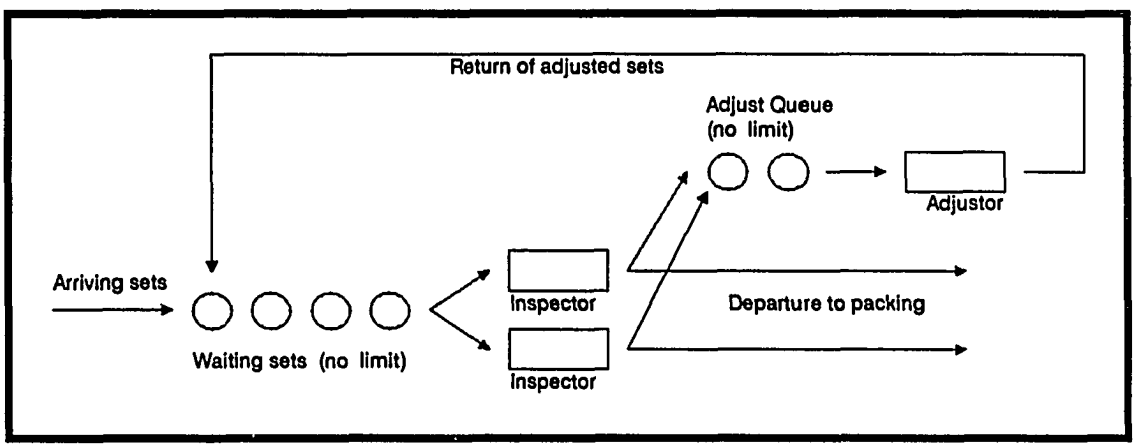

Figure 5-7. TV inspection and adjustment system

```
Simulation: TEST3      Description: TV Inspect & Adjust          Maximum Time: 480.00

Routing:    R1    Desc: Entity Creation          Ent Type: 1.0   Current Location:
   Distribution: UNFRM   Mean: 5.50  Range or Std Dev: 2.00      Failure Percent: 0.00
   Failures Go To:          Successes Go To: INSPQ              Balks Go To:
Routing:    R2    Desc: Enter Inspection Stat.  Ent Type: 1.0   Current Location: INSPQ
   Distribution:         Mean: 0.00  Range or Std Dev: 0.00      Failure Percent: 0.00
   Failures Go To:          Successes Go To: INSP               Balks Go To:
Routing:    R3    Desc: Get Inspected            Ent Type: 1.0   Current Location: INSP
   Distribution: UNFRM   Mean: 9.00  Range or Std Dev: 3.00      Failure Percent: 15.00
   Failures Go To: ADJTQ    Successes Go To:                    Balks Go To:
Routing:    R4    Desc: Enter Adjustment Stat.  Ent Type: 1.0   Current Location:ADJTQ
   Distribution:         Mean: 0.00  Range or Std Dev: 0.00      Failure Percent: 0.00
   Failures Go To:          Successes Go To: ADJT               Balks Go To:
Routing:    R5    Desc: Get Adjusted             Ent Type: 1.0   Current Location: ADJT
   Distribution: UNFRM   Mean: 30.00 Range or Std Dev: 10.00     Failure Percent: 0.00
   Failures Go To:          Successes Go To: INSPQ              Balks Go To:

Server/Queue:  INSPQ   Description: Inspection Queue    Server/Queue Cap: 999999.00
Server/Queue:  INSP    Description: Inspection          Server/Queue Cap: 2.00
Server/Queue:  ADJTQ   Description: Adjustment Queue    Server/Queue Cap: 999999.00
Server/Queue:  ADJT    Description: Adjustment          Server/Queue Cap: 1.00
```

Figure 5-8. Object data for TV inspect & adjust model

```
GEN,DIESCH,TV INSP. AND ADJUST.,1/24/89,1;
LIMITS,2,2,50;
NETWORK;
        CREATE,UNFRM(3.5,7.5),,1;          CREATE TELEVISIONS
INSP    QUEUE(1);                          INSPECTION QUEUE
        ACT(2)/1,UNFRM(6.,12.);            INSPECTION
        GOON;
        ACT,,.85,DPRT;                     85% DEPART
        ACT,,.15,ADJT;                     15% ARE RE-ADJUSTED
ADJT    QUEUE(2);                          ADJUST QUEUE
        ACT/2,UNFRM(20.,40.),,INSP;        ADJUSTMENT
DPRT    COLCT,INT(1),TIME IN SYSTEM;       COLLECT STATISTICS
        TERM;
        END;
INIT,0,480;
FIN;
```

Figure 5-9. SLAM statements for TV inspection and adjustment

Table 5-3. Simulation results for inspection and adjustment

	SLAM	SOOP
Total throughput of inspection	101	99
Average contents of inspection station	1.96	1.98
Average utilization of inspectors	0.98	0.99
Average length of adjustment queue	1.27	1.29
Total units to adjustment	15	12
Total new arrivals	86	85
Minimum time in system	6.18	6.42
Maximum time in system	140.00	184.67
Mean time in system	26.50	29.06

obtained from SLAM and the object-oriented system are similar. The next section demonstrates the simulation of a more complex system and further verifies the integrity of the object-oriented simulation program.

4. An advanced simulation model

The previous simulation models served to verify that the object-oriented simulation program is capable of performing correct simulated analyses. The model described in this section shows that the object-oriented simulation program is capable of modeling more complex systems.

One of the advantages of object-oriented programming is the modularity of program design. This modularity carries over into the use of the software. Complex simulations can be constructed through the combination of simpler models. The system shown in Figure 5-10 is a combination of the maintenance facility and the TV inspection and adjustment models.

Note that the model of Figure 5-10 represents the attachment of the output from the maintenance facility to the input of two TV inspection and adjustment

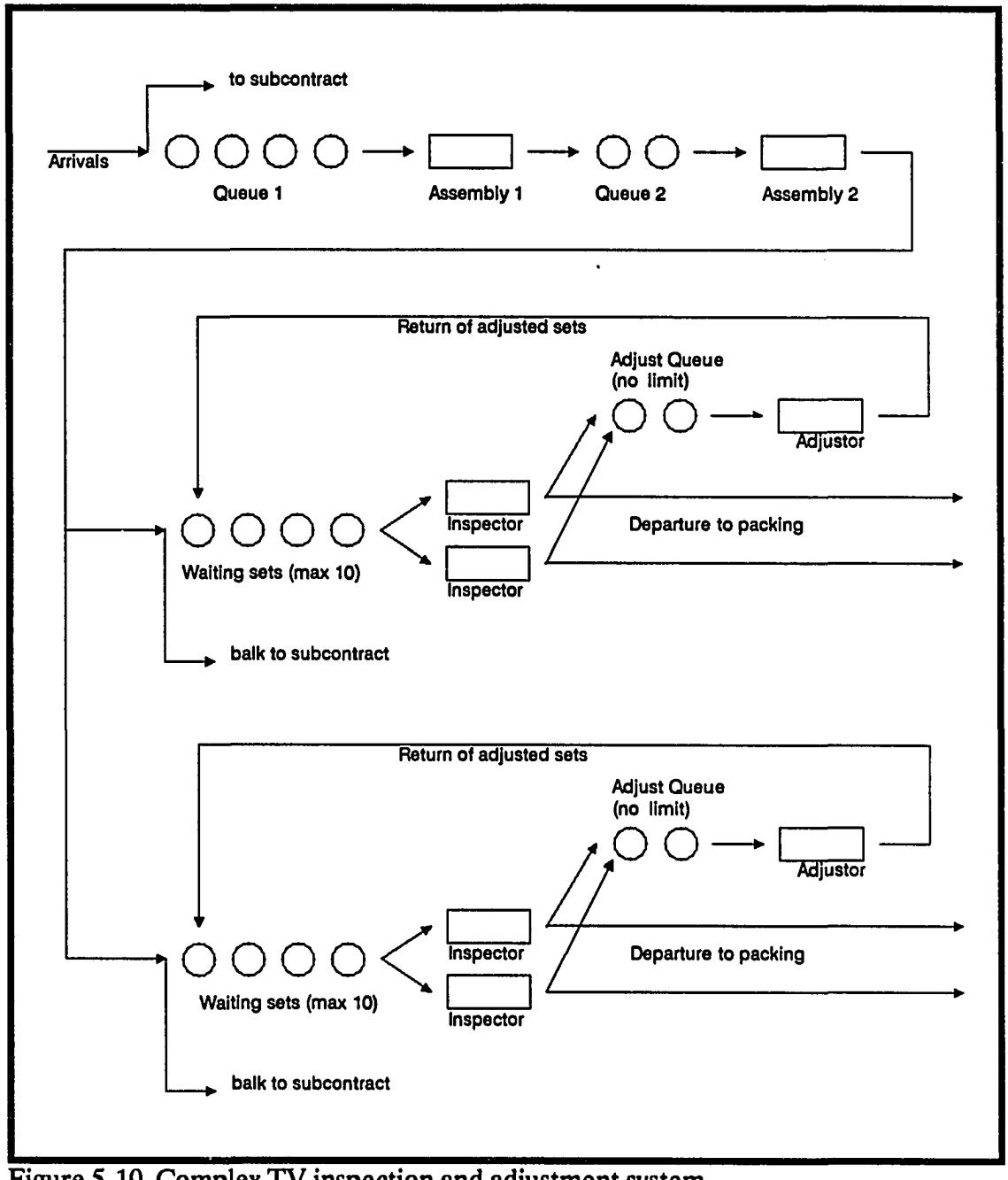

Figure 5-10. Complex TV inspection and adjustment system

Simulation: TEST4 **Description:** Combination Facility **Maximum Time:** 480.00

Routing: R1 **Desc:** Entity Creation **Ent Type:** 1.0 **Current Location:**
Distribution: EXPON **Mean:** 0.40 **Range or Std Dev:** 0.00 **Failure Percent:** 0.00
Failures Go To: **Successes Go To:** Q1 **Balks Go To:** SUBC

Routing: R2 **Desc:** Enter Service 1 **Ent Type:** 1.0 **Current Location:** Q1
Distribution: **Mean:** 0.00 **Range or Std Dev:** 0.00 **Failure Percent:** 0.00
Failures Go To: **Successes Go To:** Q2 **Balks Go To:**

Routing: R3 **Desc:** Finish Service 1 **Ent Type:** 1.0 **Current Location:** S1
Distribution: EXPON **Mean:** 0.25 **Range or Std Dev:** 0.00 **Failure Percent:** 0.00
Failures Go To: **Successes Go To:** Q2 **Balks Go To:**

Routing: R4 **Desc:** Enter Service 2 **Ent Type:** 1.0 **Current Location:**Q2
Distribution: **Mean:** 0.00 **Range or Std Dev:** 0.00 **Failure Percent:** 0.00
Failures Go To: **Successes Go To:** S2 **Balks Go To:**

Routing: R5 **Desc:** Finish Service 2 **Ent Type:** 1.0 **Current Location:** S2
Distribution: EXPON **Mean:** 0.50 **Range or Std Dev:** 0.00 **Failure Percent:** 50.00
Failures Go To: IQ2 **Successes Go To:** IQ1 **Balks Go To:**

Routing: R6 **Desc:** Finish Subcontract **Ent Type:** 1.0 **Current Location:** SUBC
Distribution: **Mean:** 0.00 **Range or Std Dev:** 0.00 **Failure Percent:** 0.00
Failures Go To: **Successes Go To:** **Balks Go To:**

Routing: R7 **Desc:** Enter Inspection **Ent Type:** 1.0 **Current Location:** INSQ1
Distribution: **Mean:** 0.00 **Range or Std Dev:** 0.00 **Failure Percent:** 0.00
Failures Go To: **Successes Go To:** INS1 **Balks Go To:**

Routing: R8 **Desc:** Inspection Station 1 **Ent Type:** 1.0 **Current Location:** INS1
Distribution: UNFRM **Mean:** 9.00 **Range or Std Dev:** 3.00 **Failure Percent:** 15.00
Failures Go To: ADJQ1 **Successes Go To:** **Balks Go To:**

Routing: R9 **Desc:** Enter Adjust1 Queue **Ent Type:** 1.0 **Current Location:** ADJQ1
Distribution: **Mean:** 0.00 **Range or Std Dev:** 0.00 **Failure Percent:** 0.00

Routing: R10 **Desc:** Adjustment Station 1 **Ent Type:** 1.0 **Current Location:** ADJ1
Distribution: UNFRM **Mean:** 30.00 **Range or Std Dev:** 10.00 **Failure Percent:** 0.00
Failures Go To: **Successes Go To:** INSQ1 **Balks Go To:** SUBI1

Routing: R11 **Desc:** Subcontract Inspect. 1 **Ent Type:** 1.0 **Current Location:** SUBI1
Distribution: **Mean:** 0.00 **Range or Std Dev:** 0.00 **Failure Percent:** 0.00
Failures Go To: **Successes Go To:** **Balks Go To:**

Routing: R12 **Desc:** Enter Inspect Queue 1 **Ent Type:** 1.0 **Current Location:** INSQ2
Distribution: **Mean:** 0.00 **Range or Std Dev:** 0.00 **Failure Percent:** 0.00
Failures Go To: **Successes Go To:** INS2 **Balks Go To:**

Routing: R13 **Desc:** Inspection Station 2 **Ent Type:** 1.0 **Current Location:** INS2
Distribution: UNFRM **Mean:** 9.00 **Range or Std Dev:** 3.00 **Failure Percent:** 15.00
Failures Go To: ADJQ2 **Successes Go To:** **Balks Go To:**

Routing: R14 **Desc:** Enter Adjust Queue 2 **Ent Type:** 1.0 **Current Location:** ADJQ2
Distribution: **Mean:** 0.00 **Range or Std Dev:** 0.00 **Failure Percent:** 0.00
Failures Go To: **Successes Go To:** ADJ2 **Balks Go To:**

Routing: R15 **Desc:** Adjustment Station 2 **Ent Type:** 1.0 **Current Location:**ADJ2
Distribution: UNFRM **Mean:** 30.00 **Range or Std Dev:** 10.00 **Failure Percent:** 0.00
Failures Go To: **Successes Go To:** INSQ2 **Balks Go To:** SUBI2

Routing: R16 **Desc:** Subcontract Inspect 1 **Ent Type:** 1.0 **Current Location:** SUBI2
Distribution: **Mean:** 0.00 **Range or Std Dev:** 0.00 **Failure Percent:** 0.00
Failures Go To: **Successes Go To:** **Balks Go To:**

Routing: R6-1 **Desc:** Temporary Queue 1 **Ent Type:** 1.0 **Current Location:** IQ1
Distribution: **Mean:** 0.00 **Range or Std Dev:** 0.00 **Failure Percent:** 0.00
Failures Go To: **Successes Go To:** INSQ1 **Balks Go To:** SUBI1

Routing: R6-2 **Desc:** Temporary Queue 2 **Ent Type:** 1.0 **Current Location:** IQ2
Distribution: **Mean:** 0.00 **Range or Std Dev:** 0.00 **Failure Percent:** 0.00
Failures Go To: **Successes Go To:** INSQ2 **Balks Go To:** SUBI2

Figure 5-11. Object data for advanced model, part 1

Server/Queue:	Q1	**Description:** Queue Number 1	**Server/Queue Cap:** 4.00
Server/Queue:	S1	**Description:** Service Number 1	**Server/Queue Cap:** 1.00
Server/Queue:	Q2	**Description:** Queue Number 2	**Server/Queue Cap:** 2.00
Server/Queue:	S2	**Description:** Service Number 2	**Server/Queue Cap:** 1.00
Server/Queue:	SUBC	**Description:** Subcontracted Parts	**Server/Queue Cap:** 1.00
Server/Queue:	INSQ1	**Description:** Inspection Queue 1	**Server/Queue Cap:** 10.00
Server/Queue:	INS1	**Description:** Inspection Station 1	**Server/Queue Cap:** 2.00
Server/Queue:	ADJQ1	**Description:** Adjust Queue 1	**Server/Queue Cap:** 999999.00
Server/Queue:	ADJ1	**Description:** Adjust Station 1	**Server/Queue Cap:** 1.00
Server/Queue:	SUBI1	**Description:** Subcontract Insp 1	**Server/Queue Cap:** 1.00
Server/Queue:	INSQ2	**Description:** Inspection Queue 2	**Server/Queue Cap:** 10.00
Server/Queue:	INS2	**Description:** Inspection Station 2	**Server/Queue Cap:** 2.00
Server/Queue:	ADJQ2	**Description:** Adjust Queue 2	**Server/Queue Cap:** 999999.00
Server/Queue:	ADJ2	**Description:** Adjust Station 2	**Server/Queue Cap:** 1.00
Server/Queue:	SUBI2	**Description:** Subcontract Insp 2	**Server/Queue Cap:** 1.00
Server/Queue:	IQ1	**Description:** Temporary Queue 1	**Server/Queue Cap:** 999999.00
Server/Queue:	IQ2	**Description:** Temporary Queue 2	**Server/Queue Cap:** 999999.00

Figure 5-12. Object data for advanced model, part 2

facilities. Interarrival times and service times remain the same. Subcontract outlets are added to the TV inspection queues and the queues have been limited to a capacity of 10 units.

The entire system is simulated for 480 time units. The data required in the objects of the object-oriented simulation program are shown in Figures 5-11 and 5-12. The SLAM statements required to simulate this system are shown in Figure 5-13. Table 5-4 shows the results of the simulations performed with both SLAM and the object-oriented simulation program. As in the three previous simulation tests, the results are similar.

In theory, there is no practical limit to the combinations that may be performed with the basic building blocks of the object-oriented simulation system. The example models serve to verify the correct operation of the object-oriented simulation program developed in this research. The next section presents a comparison between SLAM and the object-oriented approach to simulation taken in this research.

```
GEN,PRITSKER,COMBINATION SYSTEM,1/24/89,1;
LIMITS,6,2,150;
NETWORK;
;
;SUBSYSTEM 1
        CREATE,EXPON(.4),,1;              CREATE ARRIVALS
        QUEUE(1),0,4,BALK(SUB1);          STATION 1 QUEUE
        ACT/1,EXPON(.25);                 STATION 1 SERVER
        QUEUE(2),0,2,BLOCK;               STATION 2 QUEUE
        ACT/2,EXPON(.50);                 STATION 2 SERVER
        GOON;
        ACT,,0.5,INS1;                    NOW GOTO INSPECTION
        ACT,,0.5,INS2;
;
;SUBSYSTEM 2
INS1    QUEUE(3),0,10,BALK(SUB2);         INSPECTION QUEUE
        ACT(2)/3,UNFRM(6.,12.);           INSPECTION
        GOON;
        ACT,,.85,DPRT;                    85% DEPART
        ACT,,.15,ADJ1;                    15% ARE RE-ADJUSTED
ADJ1    QUEUE(4);                         ADJUST QUEUE
        ACT/4,UNFRM(20.,40.),,INS1;       ADJUSTMENT
;
;SUBSYSTEM 3
INS2    QUEUE(5),0,10,BALK(SUB3);         INSPECTION QUEUE
        ACT(2)/5,UNFRM(6.,12.);           INSPECTION
        GOON;
        ACT,,.85,DPRT;                    85% DEPART
        ACT,,.15,ADJ2;                    15% ARE RE-ADJUSTED
ADJ2    QUEUE(6);                         ADJUST QUEUE
        ACT/6,UNFRM(20.,40.),,INS2;       ADJUSTMENT
;
DPRT    COLCT,INT(1),TIME IN SYSTEM;      COLLECT STATISTICS
        TERM;
;
;STATISTICS COLLECTION ROUTINES
SUB1    COLCT,BET,TIME BET. BALKS 1;      COLLECT STATISTICS
        TERM;
SUB2    COLCT,BET,TIME BET. BALKS 2;      COLLECT STATISTICS
        TERM;
SUB3    COLCT,BET,TIME BET. BALKS 3;      COLLECT STATISTICS
        TERM;
        END
INIT,0,480;
FIN;
```

Figure 5-13. SLAM statements for advanced model

Table 5-4. Simulation results for advanced model

	SLAM	SOOP
Total throughput	1080	1167
Average length of queue 1	1.93	2.69
Average wait time in queue 1	1.04	1.27
Average length of queue 2	1.37	1.37
Average wait time in queue 2	0.74	0.82
Throughput of assembly 1	888	878
Throughput of assembly 2	885	877
Subcontracted before assembly 1	225	324
Average length of inspection queue 1	9.54	9.05
Average wait time in inspection queue 1	38.18	34.15
Average length of inspection queue 2	9.68	9.00
Average wait time in inspection queue 2	40.41	35.84
Throughput of inspection station 1	108	108
Throughput of inspection station 2	103	108
Average length of adjustment queue 1	0.12	2.36
Average wait time in adjustment queue 1	5.86	25.44
Average length of adjustment queue 2	1.36	1.89
Average wait time in adjustment queue 2	43.49	44.93
Throughput of adjustment station 1	14	12
Throughput of adjustment station 2	15	12
Subcontracted before inspection station 1	307	330
Subcontracted before inspection station 2	362	337

C. Object-oriented Versus Traditional Simulation

The simulations of the example models indicate that the end results of traditional simulation are similar to the results obtained with the object-oriented simulation program developed in this research. Assuming the correct construction and operation of the object-oriented simulation program, comparative results are to be expected. Differences between the two methods are an important component of this research. This section presents a comparison of the two approaches to simulation.

One difference of note is the execution time of the two simulation programs. The execution times for the first three example simulations were relatively close. The execution time for the advanced simulation model with SLAM was approximately 5 minutes. The same model simulated with the object-oriented simulation program took approximately 40 minutes. Such a large discrepancy in execution times presents a problem if object-oriented simulation is to be used in practice.

It is likely that the large execution time for object-oriented simulation with the software developed in this research is due to inefficiencies in the algorithms. Concentration in this research was on the correct operation of the software, not the efficient operation of the program. Careful construction of the program with less concentration on user displays would greatly enhance the operational speed of the program. The type of software development to achieve optimum performance is costly and beyond the scope of this research.

While execution appears to be slower with the object-oriented software, the measure of program execution time was made under nonvarying simulation conditions. An advantage of the object-oriented simulation program is in the interruptable nature of the program. If the user desires to modify the conditions under which the simulation is performed, a simple command may be issued to interrupt the object-oriented simulation while it is in progress. Changes may then be made to the simulation characteristics and the simulation may be restarted. With SLAM, the user must wait for the completion of the current simulation, modify the SLAM statements, and rerun the simulation. Intermediate results would be more difficult to collect. The total execution time under these conditions could easily be higher than the time required for the object-oriented simulation.

An important advantage of object-oriented programming is the ease at which the underlying program can be enhanced to provide new capabilities. For example, to add a new distribution to the simulation program a short section of code is added to the routing class method that generates samples from probability distributions. The code is then ready to draw from the new distribution. Messages used internally by the software remain unchanged. The user need only modify the distribution specified in the routing class data entry screen to use the new simulation.

The addition of a new type of probability distribution in SLAM would require the addition of code to sample from the desired distribution as well as the addition of code to correctly recognize the request for the new distribution in the SLAM statements presented to the SLAM input translation program.

The differences between the two types of simulation software are most apparent at the source code level. Screen displays, report formats, statistics collection, and general simulation capabilities are more easily modified or extended under the modular construction found in object-oriented systems.

VI. CONCLUSIONS AND RECOMMENDATIONS

Several potential benefits may be derived from this research. Through the use of object-oriented programming, simulation might be made applicable to a larger base of simulation users in business and industry. More efficient planning and better utilization of existing facilities would result in increased productivity.

An object-oriented simulation language might also better serve as an educational tool for college level courses in simulation. With object-oriented simulation, students could concentrate on the science of simulation rather than the science of programming. The proliferation of simulation into the public sector would increase as more students are educated in this area.

The primary goal of this research is to develop object-oriented extensions for simulation in a strongly-typed procedural language. This research provides a base from which future simulation languages may be built. The software industry is turning toward object-oriented programming environments for operating systems and many end-user programs. As multiprocessor, multitasking computers become readily available, the object-oriented approach taken in this research will provide an efficient vehicle for simulation program design. The inherent characteristics of the object-oriented programming paradigm fit well with the parallel process architecture that will be a part of the future of computing.

The result of the research is an object-oriented simulation language that may be used in industry and classroom settings. At best, the simulation of a real-world system should be constructed using the same terminology, methods, and skills required to construct the real system. Most computer languages operate under the "data-procedure" paradigm. Procedures (distinct sections of computer code) act on data passed to them. Procedures must be prepared for every type of task required

by the resultant program. Object-oriented languages employ a data or "object-oriented" approach to programming. Instead of passing data to procedures, the data (objects) perform operations on themselves.

Further, in object-oriented programming, the types of operations performed on data can be developed in an abstract way so that a separate procedure is not required for each operation. Instead, a "class" structure is used. A single class provides all the information necessary to construct and use objects of a particular kind (instances of a class). All operations on objects are carried out by passing "messages" to an instance of a class. Messages in an object-oriented language occur simultaneously and are automatically passed from class to class.

The object-oriented approach allows straightforward simulation modeling by removing the simulation expert from the process. Little training in simulation methods is necessary because the expertise required is already available through the persons working with the real-world system.

Computers and computer simulation will become more complex in the future. The advantages of object-oriented simulation will facilitate future simulation research. Future research in object-oriented simulation should concentrate on the optimization of methods used to implement the simulation functions. One area of concentration could be the intelligent selection of messages from the message queue when the queue contains messages of equal priority. Currently, the messages are scanned in turn until an action can be taken. Intelligent selection would allow only messages that are ready for execution to be retrieved from the message queue. Other code segments could be optimized with assembly language subroutines. Another area of future research is the investigation of object-oriented operating systems to serve as the basis for object-oriented simulation programs.

V. BIBLIOGRAPHY

1. Arthur, J. L.; Frendewey, James O.; Ghandforoush, Parviz; and Rees, Loren Paul. "Microcomputer Simulation Systems". Computers and Operations Research 13 (February 1986): 167-183.

2. Banks, Jerry; and Carson, John S. II. "Process-interaction Simulation Languages". Simulation 44 (May 1985): 225-235.

3. Barnett, Claude C. "Micro PASSIM: A Modeling Package for Combined Simulation using Turbo Pascal". Proceedings of the Conference on Modeling and Simulation on Microcomputers (1985): 37-41.

4. Barnett, Claude C. "MICRO-PASSIM: A Combined Simulation Package for a Microcomputer using UCSD Pascal". Proceedings of the Conference on Modeling and Simulation on Microcomputers (1983): 92-95.

5. Barta, Thomas A. "Animated Simulation Graphics with GPSS". Proceedings of the Conference on Modeling and Simulation on Microcomputers (1985): 51-54.

6. Bell, Peter C.; and O'Keefe, Robert M. "Visual Interactive Simulation - History, Recent Developments, and Major Issues". Simulation 49 (September 1987): 109-116.

7. Bezevin, Jean. "Timelock: A Concurrent Simulation Technique and its Description in Smalltalk-80". 1987 Winter Simulation Conference Proceedings (1987): 503-506.

8. Birtwistle, Graham. "Future Directions in Simulation Software (Panel)". Proceedings of the Conference on Simulation in Strongly Typed Languages 13 (February 1984): 120-121.

9. Birtwistle, Graham; Wyvill, Brian; Levinson, Danny; and Neal, Radford. "Visualising a Simulation Using Animated Pictures". Proceedings of the Conference on Simulation in Strongly Typed Languages 13 (February 1984): 57-61.

10. Cammarata, Stephanie; Gates, Barbara; and Rothenberg, Jeff. "Dependencies and Graphical Interfaces in Object-Oriented Simulation Languages". 1987 Winter Simulation Conference Proceedings (1987): 507-517.

11. Carroll, John M. Simulation Using Personal Computers. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1987.

12. Cobbin, Philip. "SIMPLE_1: A Simulation Environment for the IBM PC". Proceedings of the Conference on Modeling and Simulation on Microcomputers (1986): 243-248.

13. Concepcion, Arturo I. "The Implementation of the Hierarchial Abstract Simulator on the HEP Compter". 1985 Winter Simulation Conference Proceedings (1985): 428-434.

14. Cornish, Merrill. "AI Corner: What Would You Do with Object-Oriented Programming if You Had It"? DirecTIons 5 (March 1988): 28-44.

15. Cox, Brad; and Hunt, Bill. "Objects, Icons, and Software-ICs". BYTE 11 (August 1986): 161-176.

16. Cox, B.J. Object-Oriented Programming: An Evolutionary Approach. Reading, MA: Addison-Wesley, 1986.

17. Cox, Springer. "Interactive Graphics in GPSS/PC". Simulation 49 (September 1987): 117-122.

18. Cox, Springer; and Cox, Alice J. "GPSS/PC: A User Oriented Simulation System". Proceedings of the Conference on Modeling and Simulation on Microcomputers (1985): 48-50.

19. Decker, H.; and Maierhofer, J. "Very High Level Model Description and Simulation". Proceedings of the Conference on Simulation in Strongly Typed Languages 13 (February 1984): 44-48.

20. Duff, Charles B. "Designing an Efficient Language". BYTE 11 (August 1986): 211-224.

21. Favreau, Romeo R.; and Marr, George R. Jr. "EzSIM - A Desktop Database System for Simulation Development and Documentation". Proceedings of the Conference on Modeling and Simulation on Microcomputers (1986): 129-133.

22. Fernhout, Paul D. "Simulating Interacting Intelligent Objects in C". AI Expert 4 (January 1989): 38-46.

23. Fisher, Edward L. "An AI-Based Methodology for Factory Design". AI Magazine 7 (Fall 1986): 72-85.

24. Frantz, Frederick K.; and Trott, Kevin C. "Extensions to Pascal for Discrete Event Simulation". Proceedings of the Conference on Simulation in Strongly Typed Languages 13 (February 1984): 32-36.

25. Golden, Donald G. "Software Engineering Considerations for the Design of Simulation Languages". Simulation 45 (October 1985): 169-178.

26. Grant, John W.; and Weiner, Steven A. "Factors to Consider in Choosing a Graphically Animated Simulation System". Industrial Engineering 18 (August 1986): 37-68.

27. Haigh, Peter L.; and Bornhorst, Ellen M. "A User Friendly Environment for Simulating Computer Systems". Proceedings of the Conference on Modeling and Simulation on Microcomputers (1986): 134-140.

28. Hollocks, Brian W. "Practical Benefits of Animated Graphics in Simulation". 1984 Winter Simulation Conference Proceedings (1984): 323-328.

29. Hoover, Stewart. "MICRO-SIM: A Simulation Package for Microcomputers". Proceedings of the Conference on Modeling and Simulation on Microcomputers (1983): 87-91.

30. Hughes, D. J. F.; and Gunadi, H. "S/Pascal: A Portable Simulation Language Based on Pascal". Proceedings of the Conference on Modeling and Simulation on Microcomputers (1984): 116-120.

31. Hurrion, R. D. "Visual Interactive Simulation Using a Microcomputer". Computers and Operations Research 8 (April 1981): 267-273.

32. Jefferson, David. "Future Directions in Simulation at the Conference on Simulation in Strongly Typed Languages (Panel)". Proceedings of the Conference on Simulation in Strongly Typed Languages 13 (February 1984): 123-124.

33. Johnson, Glen D.; Rector, Brent E.; and Mullarney, Alasdar. "Tabletop SIMSCRIPT". Proceedings of the Conference on Modeling and Simulation on Microcomputers (1983): 96-102.

34. Johnson, M. Eric; and Poorte, Jacob P. "A Hierarchical Approach to Computer Animation in Simulation Modeling". Simulation 50 (January 1988): 30-36.

35. Kaehler, Ted; and Patterson, Dave. "A Small Taste of Smalltalk". BYTE 11 (August 1986): 145-158.

36. Karian, Zaven A.; and Dudewicz, Edward J. "Discrete-event Simulation on Microcomputers". Proceedings of the Conference on Modeling and Simulation on Microcomputers (1986): 146-150.

37. King, Christina U.; and Fisher, Edward L. "Object-Oriented Shop-Floor Design, Simulation, and Evaluation". Fall Industrial Engineering Conference Proceedings (1986): 131-137.

38. Knapp, Verna. "The Smalltalk Simulation Environment". 1986 Winter Simulation Conference Proceedings (1986): 125-128.

39. Knapp, Verna. "The Smalltalk Simulation Environment, Part II". 1987 Winter Simulation Conference Proceedings (1987): 146-151.

40. Kootsey, J. Mailen; and Holt, Donald C. "A General-purpose Interactive Control Program for Simulations on Microcomputers". Proceedings of the Conference on Modeling and Simulation on Microcomputers (1984): 112-115.

41. Langlois, Laurent. "Simulation Visualization with SIMSEA, a General Purpose Animation Language". Proceedings of the Conference on Simulation in Strongly Typed Languages 13 (February 1984): 62-67.

42. L'Ecuyer, Pierre; and Giroux, Nataly. "A Process-Oriented Simulation Package Based On Modula-2". 1987 Winter Simulation Conference Proceedings (1987): 165-174.

43. Levine, Robert I.; Drang, Diane E.; and Edelson, Barry. A Comprehensive Guide to AI and Expert Systems. New York, New York: McGraw-Hill, 1986.

44. Macintosh, J. B.; Hawkins, R. W.; and Sheppard, C. J. "Simulation on Microcomputers - The Development of a Visual Interactive Modeling Philosophy". 1984 Winter Simulation Conference Proceedings (1984): 531-537.

45. MacLennan, Bruce J. Principles of Programming Languages. New York, New York: CBS College Publishing, 1987.

46. Magnenat-Thalmann, Nadia; and Thalmann, David. "Procedural Animation Blocks in Discrete Simulation". Simulation 49 (September 1987): 102-108.

47. Magnenat-Thalmann, N; and Thalmann, D. "Animated Types and Actor Types in Computer Simulation and Animation". Proceedings of the Conference on Simulation in Strongly Typed Languages 13 (February 1984): 51-56.

48. Malloy, Brian; and Soffa, Mary Lou. "SIMCAL: The Merger of Simula and Pascal". 1986 Winter Simulation Conference Proceedings (1986): 397-403.

49. Marr, George R., Jr. "SIM_BY_INT - The Next Generation Simulation Language". Proceedings of the Conference on Modeling and Simulation on Microcomputers (1984): 121-123.

50. Mathewson, S.C. 'The Application of Program Generator Software and Its Extensions to Discrete Event Simulation Modeling". IIE Transactions 16 (1984): 3-18.

51. McFall, Michael E.; and Klahr, Philip. "Simulation With Rules and Objects". 1986 Winter Simulation Conference Proceedings (1986): 470-473.

52. Meerman, J. W. "Dynamic Systems Simulation with Personal Computers and TUTSIM". Proceedings of the Conference on Modeling and Simulation on Microcomputers (1983): 106-111.

53. Nance, Richard E. "Simulation Modeling: Two Perspectives". IIE Transactions 16 (1984): 2.

54. O'Keefe, Robert. "Simulation and Expert Systems - A Taxonomy and Some Examples". Simulation 46 (January 1986): 10-16.

55. Pascoe, Geoffrey A. "Elements of Object-Oriented Programming". BYTE 11 (August 1986): 139-144.

56. Pountain, Dick. "Object-Oriented Forth". BYTE 11 (August 1986): 227-233.

57. Pratt, Charles A. "Catalog of Simulation Software". Simulation 49 (October 1987): 165-181.

58. Pritsker, Alan B. Introduction to Simulation and Slam II. New York, New York: John Wiley & Sons, Inc., 1986.

59. Reddy, Y. V. Ramana; Fox, Mark S.; Husain, Nizwer; and McRoberts, Malcolm. "The Knowledge-Based Simulation System". IEEE Software 3 (March 1986): 26-37.

60. Rothenberg, Jeff. "Object-Oriented Simulation: Where Do We Go from Here"? 1986 Winter Simulation Conference Proceedings (1986): 464-469.

61. Rozenblit, Jerzy W.; and Zeigler, Bernard P. "Concepts for Knowledge-Based System Design Environments". 1985 Winter Simulation Conference Proceedings (1985): 223-231.

62. Rozenblit, Jerzy W.; Sevinc, Suleyman; and Zeigler, Bernard P. "Knowledge-Based Design of LANs Using System Entity Structure Concepts". 1986 Winter Simulation Conference Proceedings (1986): 858-865.

63. Ruiz-Mier, Sergio; Talavage, Joseph; and Ben-Arieh, David. "Towards a Knowledge-Based Network Simulation Environment". 1985 Winter Simulation Conference Proceedings (1985): 232-236.

64. Samuels, Michael L.; and Spiegel, James R. "The Flexible Ada Simulation Tool (FAST) and its Extensions". 1987 Winter Simulation Conference Proceedings (1987): 175-184.

65. Saydam, Tuncay. "Process-Oriented Simulation Languages". Simuletter 16 (April 1985): 8-13.

66. Schriber, Thomas J. Simulation Using GPSS. New York, New York: John Wiley & Sons, Inc., 1974.

67. Schwetman, Herb. "CSIM: A C-based, Process-Oriented Simulation Language". 1986 Winter Simulation Conference Proceedings (1986): 387-396.

68. Seila, Andrew F. "SIMTOOLS: A Software Tool Kit for Discrete Event Simulation in Pascal". Simulation 50 (March 1988): 93-99.

69. Shanehchi, J. "EXPRESS: A Man-machine Interface for Simulation". Proceedings of the 1st International Conference on Simulation in Manufacturing 1 (1985): 97-105.

70. Smith, B. J.; and Smith, M. Z. "The Pascal/VS Simulation Tool: Overview and Examples". Proceedings of the Conference on Simulation in Strongly Typed Languages 13 (February 1984): 27-31.

71. Smith, Richard L.; and Platt, Lucille. "Benefits of Animation in the Simulation of a Machining and Assembly Line". Simulation 48 (January 1987): 28-30.

72. Stairmand, Malcolm C.; and Kreutzer, Wolfgang. "POSE: A Process-Oriented Simulation Environment Embedded in SCHEME". Simulation 50 (April 1988): 143-153.

73. Standridge, Charles R. "Performing Simulation Project with The Extended Simulation System (TESS)". Simulation 45 (December 1985): 283-291.

74. Stein, Jacob. "Object-Oriented Programming and Databases". Dr. Dobb's Journal 13 (March 1988): 18-34.

75. Taha, Hamdy A. Simulation Modeling and SIMNET. Englewood Cliffs, New Jersey: Prentice Hall, Inc., 1988.

76. Tesler, Larry. "Programming Experiences". BYTE 11 (August 1986): 195-206.

77. Thesen, Arne. "Writing Simulations from Scratch: Pascal Implementations". 1987 Winter Simulation Conference Proceedings (1987): 152-164.

78. Ulgen, Onur M.; and Thomasma, Timothy. "Simulation Modeling in an Object-Oriented Environment Using Smalltalk-80". 1986 Winter Simulation Conference Proceedings (1986): 474-484.

79. Unger, Brian W. "Object Oriented Simulation". 1986 Winter Simulation Conference Proceedings (1986): 123-124.

80. Vaucher, Jean. "Future Directions in Simulation Software (Panel)". Proceedings of the Conference on Simulation in Strongly Typed Languages 13 (February 1984): 122.

81. Vaucher, Jean. "Process-oriented Simulation in Standard Pascal". Proceedings of the Conference on Simulation in Strongly Typed Languages 13 (February 1984): 37-43.

82. Wadsworth, Richard B. "MICRO-PASSIM with Graphics". Proceedings of the Conference on Modeling and Simulation on Microcomputers (1983): 103-105.

83. Zeigler, Bernard P. "Hierarchical Modular Modeling/Knowledge Representation". 1986 Winter Simulation Conference Proceedings (1986): 129-137.

84. Zeigler, Bernard P. "Hierarchical, Modular Discrete-event Modeling in an Object-oriented Environment". Simulation 49 (November 1987): 219-230.

85. Zeigler, Bernard P. "System-Theoretic Representation of Simulation Models". IIE Transactions 16 (1984): 19-34.

86. Zeigler, Bernard P.; and Kim, Tag Gon. "The DEVS Formalism: Hierarchical, Modular Systems Specification in an Object Oriented Framework". 1987 Winter Simulation Conference Proceedings (1987): 559-566.

VIII. APPENDIX. SIMULATION SOFTWARE SOURCE CODE

```
{ SOOP.PAS }

{$M 16384,16384,655360} { stacksize,heapmin,heapmax }

program SOOP;
{////////////////////////////////////////////////////////////
//                                                          //
//   Program: Simulation with Object Oriented Programming   //
//   Version: 1.0                                           //
//   Revised: 1/89                                          //
//                                                          //
//   Prepared by:      Kurt H. Diesch                       //
//                                                          //
//   Simulation with object-oriented programming (SOOP)     //
//                                                          //
////////////////////////////////////////////////////////////}

{$I COMPDIRS.PAS }                  { compiler directives }

uses SOOPMSG; { message handling unit, the only unit aware of
                all classes! }

begin
  MessageHandler;   { branch to Message Handler }
end.


{COMPDIRS.PAS}

{$B+}      { full boolean evaluation (always on) }
{$F+}      { force far calls }
{$I-}      { I/O Checking off }
{$R-}      { range checking off }
{$S+}      { Stack checking off }
{$V-}      { Var-string checking (always off) }
```

```
{ SOOPDEFS.PAS }

{============= STANDARD DEFINITIONS ====================}

const
  BackC   : byte = $01;   { background color (Blue)         }
  LowC    : byte = $13;   { low color       (Cyan on Blue)  }
  NormC   : byte = $1F;   { norm color      (White on Blue) }
  InvC    : byte = $31;   { inverse color   (Blue on Cyan)  }
  HeadC   : byte = $1E;   { headline color (Yellow on Blue) }
  ErrorC  : byte = $4F;   { error color     (White on Red)  }
  HelpC   : byte = $3E;   { headline color (Green on Blue)  }
  DefBeep : boolean = FALSE;              { computer beeper }

type
  MenuSet    = Set of Byte;
  TimeStr    = string[6];
  DateStr    = string[10];
  Str12      = string[12];
  HexStr     = string[2];
  LineArray  = array [1..160] of byte;

  Command    = record          { menu strings }
                 Line : array [1..2] of string;
                 Desc : array [1..25] of string[80];
               end;

  WindowPtr = ^WindowArray;
  WindowArray= record          { space for saving screens }
                 Add: array [0..24,0..79] of word;
                 ULX,ULY,LRX,LRY : byte;
               end;

const

  NOKEY=0; BACK=8; CR=13; ESC=27; SPACE=32;   { keystrokes }
  F1=187; F2=188; F3=189; F4=190; F5=191; F6=192; F7=193;
  F8=194; F9=195; F10=196;
  HOME=199; ENDKEY=207; PGUP=201; PGDN=209; CTRLPGDN=246;
  UP=200; LEFT=203; RIGHT=205; DOWN=208;
  INSKEY=210; DELKEY=211; RTAB=9; LTAB=143;
  CTRLLEFT=243; CTRLRIGHT=244; CTRLEND=245; CTRLHOME=247;

  PLF  = #10;                  { print code mnemonics }
  PCR  = #13;
  PCRLF= #13#10;
  PFF  = #13#12;
```

```
  CBack     = #255#0;          { activates background color }
  CLow      = #255#1;          { activates low color }
  CNorm     = #255#2;          { activates normal color }
  CInv      = #255#3;          { activates inverse color }
  CHead     = #255#4;          { activates headline color }
  CError    = #255#5;          { activates error color }
  CHelp     = #255#6;          { activates help line color }

  CursorOn  : word  = $0607;  { default cursor }
  CursorBlk : word  = $0507;  { default block cursor }
  CursorOff : word  = $2020;  { cursor off value }

  EMPTYSET  : MenuSet = [];   { the empty set }
  ALLCHAR   : MenuSet = [32..126];    { all printable set }
  INTS      : MenuSet = [45,48..57];  { integer input set }
  REALS     : MenuSet = [45,46,48..57];  { real input set }
  WORDS     : MenuSet = [48..57];     { integer input set }
  YESNO     : MenuSet = [78,110,89,121];     { yes/no set }
  YES       : MenuSet = [89,121];              { yes set }
  NO        : MenuSet = [78,110];               { no set }
  HEXES     : MenuSet = [48..57,65..70,97..102];{ hex set }
  TIMESET   : MenuSet = [48..57,65,80,97,112]; { time set }
  FILECHAR  : MenuSet =
              [33,35..41,45,48..57,64..90,96..123,125,126];

  MENULINE   =  22;     { line to show menus }
  MSGLINE    =  24;     { line to show messages }
  HELPLINE   =  25;     { line to show help }
  MAXCOMLIST = 127;     { maximum # of command lists }

  MinMem     = $80;     { minimum allowable memory }

var
  ScreenAdr    : word;         { screen address }
  RetraceMode  : boolean;      { wait for retrace? }
  IsMono       : boolean;      { is this a mono monitor? }
  OrigTextAt   : byte;         { original text attribute }
  SavedExitProc: pointer;      { old ExitProc value }
  OldInt24     : pointer;      { old Int24 vector }
  DosBreakState: boolean;      { Initial state of DOS break }
  CritError    : word;         { critical error number }
  PASError     : word;         { error number }
  AMSTError    : word;         { global error number }
  CurrentCursor: byte;         { current cursor mode }
  CmdList      : integer;      { current command list }
  CmdNum       : array [1..MaxComList] of byte;
  Commands     : Command;      { command list detail }
  CurrCommand  : integer;      { current command to execute }
  VList        : array [0..20] of string[40];
  AMSTTop      : ^word;        { top of heap }
  OldScreen    : WindowPtr;    { screen storage }


{============= OBJECT CLASS DECLARATIONS ==================}


const
  MaxClasses = 4;    { maximum number of object classes }
  MAILMAN  = 0;    { mailman (messenger) class }
  SIMULATE = 1;    { simulation class }
  ENTITY   = 2;    { entity class }
  ROUTING  = 3;    { routing class }
  SERVQUE  = 4;    { service/queue class }

  CLSNAMES : array [0..MaxClasses] of string[8] =
     ('MAILMAN','SIMULATE','ENTITY','ROUTING','SERVQUE');

  CASES     : MenuSet = [69,101,76,108,85,117];
  KEYSET    : MenuSet = [68,100,78,110,85,117];
  FTYPESET  : MenuSet =
                [65..69,72,73,78,80,82..84,87,89,97..101,
                 104,105,110,112,114..116,119,121];

  DBFORMLEN  = 50;                  { max formula length }
  DBTITLELEN = 10;                  { max title length }

type
  DBTitleStr = string[DBTITLELEN];  { title type string }
  DBFormStr  = string[DBFORMLEN];   { formula string type }

const
  DBMAXRECLEN =  500;   { maximum size of a record }
  DBMINRECLEN =   14;   { minimum size of a record }
  DBMAXFIELDS =  100;   { maximum number of database fields }
  DBMAXFLDLEN =   75;   { maximum field length }
  DBMINY      =    3;   { min Y screen position }
  DBMAXY      =   20;   { max Y screen position }
```

```
  DBBBYTE : byte       = 0;      { field defaults }
  DBBCHAR : char       = ' ';
  DBBENTRY: string[6]  = '000000';
  DBBINT  : integer    = 0;
  DBBREAL : real       = 0.0;
  DBBWORD : word       = 0;
  DBBMIN  : byte       = 0;      { allowable numeric ranges }
  DBBMAX  : byte       = 255;
  DBIMIN  : integer    = -32768;
  DBIMAX  : integer    = 32767;
  DBRMIN  : real       = -9.9999999999E+14;
  DBRMAX  : real       =  9.9999999999E+14;
  DBWMIN  : word       = 0;
  DBWMAX  : word       = 65535;
  DBCALC  : boolean = TRUE;      { field def identifiers }
  DBNCALC : boolean = FALSE;
  DBMAND  : boolean = TRUE;
  DBNMAND : boolean = FALSE;
  DBWTITLE: boolean = TRUE;
  DBNTITLE: boolean = FALSE;
  DBUPLOW : char    = 'E';
  DBLOWC  : char    = 'L';
  DBUPC   : char    = 'U';
  DBNKEY  : char    = 'N';

type
  DBFieldPtr = ^DBField;
  DBField  = record  { input screen definition }
     Title : DBTitleStr;    { field title            }
     FType : char;          { field type             }
     Len   : byte;          { field length           }
     Decs  : byte;          { decimal precision      }
     X     : byte;          { X position             }
     Y     : byte;          { Y position             }
     Page  : byte;          { field page             }
     ALen  : byte;          { byte length of field   }
     AOfs  : integer;       { offset into record     }
     CCase : char;          { up/low conversion type }
     Mand  : boolean;       { mandatory entry?       }
     Calc  : boolean;       { calculated field       }
     KType : char;          { key: N)o D)ups U)nique }
     OkSet : MenuSet;       { allowable chars        }
     Form  : DBFormStr;     { formula for this field }
     WTitle: boolean;       { on screen w/title?     }
  end;
```

```
  DBScrLineRec = record     { screen line record }
                   Page : byte;
                   Line : byte;
                   Cont : array [0..79] of word;
                 end;

  DBFileRec = record      { database definition file record }
                case RType: byte of
                  0: ( FieldDef: DBField );
                  1: ( ScrLine : DBScrLineRec );
              end;

  DBBufPtr   = ^DBBufArray;
  DBBufArray = array [0..DBMAXRECLEN] of byte;   { buffer }

  DBFDataArray = array [0..DBMAXFLDLEN] of byte; { buffer }

  DBFieldArray = array [0..DBMAXFIELDS] of DBFieldPtr;



{============= MESSAGE TYPE DECLARATIONS ====================}

type

  InstType   = string[5];  { instance identifier type }

  StatusType = (IDLE,BUSY,BLOCKED);  { server status }

  MsgType    = (

    NMSG,          { nil message }

    CLEAR_OBJ,     { clear object data fields }
    DELETE_OBJ,    { delete an instance of an object }
    ENTER_OBJ,     { enter (user) new data for an object }
    LOAD_OBJ,      { load simulation objects from disk }
    SAVE_OBJ,      { save simulation objects to disk }
    SHOW_CURR_OBJ, { show current instance of an object }
    SHOW_NEXT_OBJ, { show next instance of an object }
    SHOW_PREV_OBJ, { show previous instance of an object }
    UPDATE_OBJ,    { update (user) the data for an object }
    UPDATE_CLOCK,  { update the simulation clock }
```

```
    GEN_ARR_TIME,  { determine arrival to generate & when }
    GEN_ARRIVAL,   { general next arrival of an entity }
    GET_NEXT_RTE,  { get next routing for an object }
    GET_ALT_RTE,   { request denied, get alternate route }
    GET_FAIL_RTE,  { request service/queue after failure }
    GET_FAIL_RTRY, { request denied after failure, retry }
    REQ_SQ_ENTRY,  { entity request for service or queue }
    REQ_SQ_GRANTED,{ request for service/queue granted }
    REQ_SQ_DENIED, { request for service/queue denied }
    REQ_SQ_COMP,   { request completion of service time }
    SCH_SQ_COMP,   { schedule the completion of service }
    SQ_COMPLETE,   { a service has been completed }
    ENTITY_SQ_COMP,{ tell entity it completed service/queue }
    ENTITY_LEAVE_SQ,  { tell service/queue entity has left }
    ENTITY_SET_FAIL,  { set an entity to fail service }
    ENTITY_NO_FAIL,   { set an entity not to fail service }
    ENTITY_DEP,       { entity has departed system }
    LEAVE_SYS,        { tell entity to leave system }

    REPORT_SIM,    { report on the simulation }
    END_SIMULATION { end the current simulation }

  );


  MsgPacketPtr  = ^MsgPacketType;
  MsgPacketType = record
                    FromCls : byte;     { from class }
                    FromInst: InstType; { from instance }
                    ToCls   : byte;     { to class }
                    ToInst  : InstType; { to instance }
                    Message : MsgType;  { the message }
                    Number  : real;     { number to pass }
                    Clock   : real;     { time to execute }
                    Next    : MsgPacketPtr; { next message }
                  end;

const

  ROUNDFACT: real = 0.1;  { rounding for service denials }
  NINST    : InstType = '     ';   { a nil instance id }
  PRIORITY : real = -1.0;          { priority message flag }
```

```
  SoopMsgs : array [0..30] of string[20] = (
    'NIL_MESSAGE',       { 0  }
    'CLEAR_OBJ',         { 1  }
    'DELETE_OBJ',        { 2  }
    'ENTER_OBJ',         { 3  }
    'LOAD_OBJ',          { 4  }
    'SAVE_OBJ',          { 5  }
    'SHOW_CURR_OBJ',     { 6  }
    'SHOW_NEXT_OBJ',     { 7  }
    'SHOW_PREV_OBJ',     { 8  }
    'UPDATE_OBJ',        { 9  }
    'UPDATE_CLOCK',      { 10 }
    'GEN_ARR_TIME',      { 11 }
    'GEN_ARRIVAL',       { 12 }
    'GET_NEXT_RTE',      { 13 }
    'GET_ALT_RTE',       { 14 }
    'GET_FAIL_RTE',      { 15 }
    'GET_FAIL_RTRY',     { 16 }
    'REQ_SQ_ENTRY',      { 17 }
    'REQ_SQ_GRANTED',    { 18 }
    'REQ_SQ_DENIED',     { 19 }
    'REQ_SQ_COMP',       { 20 }
    'SCH_SQ_COMP',       { 21 }
    'SQ_COMPLETE',       { 22 }
    'ENTITY_SQ_COMP',    { 23 }
    'ENTITY_LEAVE_SQ',   { 24 }
    'ENTITY_SET_FAIL',   { 25 }
    'ENTITY_NO_FAIL',    { 26 }
    'ENTITY_DEP',        { 27 }
    'LEAVE_SYS',         { 28 }
    'REPORT_SIM',        { 29 }
    'END_SIMULATION'     { 30 }

  );

var
  FirstMsg : MsgPacketPtr;  { working pointer for messages }
  SimName  : string[8];     { name of current simulation }
  CurrCls  : byte;          { currently displayed class }
  SimClock : real;          { current simulation time   }
  Paused   : boolean;       { is simulation paused? }
  SStep    : boolean;       { is single stepping on? }
  MsgCount : word;          { current count of messages }
```

```
;       AMSTSCRN.ASM
;       Fast screen writing routines

DATA    SEGMENT BYTE PUBLIC

        EXTRN   ScreenAdr:WORD              ;Pascal variables
        EXTRN   RetraceMode:BYTE

DATA    ENDS

CODE    SEGMENT BYTE PUBLIC

        ASSUME  CS:CODE,DS:DATA

        PUBLIC  FastWrite,ChangeAttribute
        PUBLIC  MoveFromScreen,MoveToScreen

;*******************************************************

;calculate Offset in video memory.
;On entry, AX has Row, DI has Column
;On exit, ES has ScreenAdr, DI has offset

CalcOffset      PROC NEAR

        DEC     AX                  ;Row to 0..24 range
        MOV     CX,50H              ;CX = Rows per column
        MUL     CX                  ;AX = Row * 80
        DEC     DI                  ;Column to 0..79 range
        ADD     DI,AX               ;DI = (Row * 80) + Col
        SHL     DI,1                ;Account for attribute
        MOV     ES,ScreenAdr        ;ES:DI points to Row,Col
        RET                         ;Return

CalcOffset      ENDP

;*******************************************************

;procedure FastWrite(St : String; Row, Col, Attr : Integer);

;Write St at Row,Col in Attr (video attribute) without snow

FWAttr          EQU     BYTE PTR [BP+6]
FWCol           EQU     WORD PTR [BP+8]
FWRow           EQU     WORD PTR [BP+10]
FWSt            EQU     DWORD PTR [BP+12]

FastWrite       PROC FAR

        PUSH    BP              ;Save BP
        MOV     BP,SP           ;Set up stack frame
        PUSH    DS              ;Save DS
        MOV     AX,FWRow        ;AX = Row
        MOV     DI,FWCol        ;DI = Column
        CALL    CalcOffset      ;calculate offset
        MOV     CL,RetraceMode  ;Grab before changing DS
        LDS     SI,FWSt         ;DS:SI points to St[0]
        CLD                     ;Set direction to forward
        XOR     AX,AX           ;AX = 0
        LODSB                   ;AX = Length(St);
        XCHG    AX,CX           ;CX = Length; AL = Wait
        JCXZ    FWExit          ;If string empty, exit
        MOV     AH,FWAttr       ;AH = Attribute
        RCR     AL,1            ;If RetraceMode is False
        JNC     FWMono          ; use "FWMono" routine
        MOV     DX,03DAh        ;point DX to CGA status
FWGetNext:
        LODSB                   ;Load next char into AL
                                ; AH already has Attr
        MOV     BX,AX           ;Store video word in BX
        CLI                     ;No interrupts now
FWWaitNoH:
        IN      AL,DX           ;Get 6845 status
        TEST    AL,8            ;Vert retrace in progress?
        JNZ     FWStore         ;If so, go
        RCR     AL,1            ;Else, wait for end of
        JC      FWWaitNoH       ; horizontal retrace
FWWaitH:
        IN      AL,DX           ;Get 6845 status again
        RCR     AL,1            ;Wait for horizontal
        JNC     FWWaitH         ; retrace
FWStore:
        MOV     AX,BX           ;Move word back to AX...
        STOSW                   ; and then to screen
        STI                     ;Allow interrupts!
        LOOP    FWGetNext       ;Get next character
        JMP     FWExit          ;Done
FWMono:
        LODSB                   ;Load next char into AL
                                ; AH already has Attr
        STOSW                   ;video word into place
        LOOP    FWMono          ;Get next character
```

```
FWExit:
        POP     DS                      ;Restore DS
        MOV     SP,BP                   ;Restore SP
        POP     BP                      ;Restore BP
        RET     10                      ;Remove parms and return

FastWrite       ENDP

;*******************************************************

;procedure ChangeAttribute(Number : Integer; Row, Col, Attr:
                           Integer);

;Change Number video attributes to Attr starting at Row,Col

CAAttr          EQU     BYTE PTR [BP+6]
CACol           EQU     WORD PTR [BP+8]
CARow           EQU     WORD PTR [BP+10]
CANumber        EQU     WORD PTR [BP+12]

ChangeAttribute PROC FAR

        PUSH    BP                      ;Save BP
        MOV     BP,SP                   ;Set up stack frame
        MOV     AX,CARow                ;AX = Row
        MOV     DI,CACol                ;DI = Column
        CALL    CalcOffset              ;calculate offset
        INC     DI                      ;Skip character
        MOV     AL,CAAttr               ;AL = Attribute
        CLD                             ;Set direction to forward
        MOV     CX,CANumber             ;CX = Number to change
        JCXZ    CAExit                  ;If zero, exit
        CMP     RetraceMode,1           ;Get wait state
        JNE     CANoWait                ;If RetraceMode is false
                                        ; use CANoWait routine
        MOV     AH,AL                   ;Store attribute in AH
        MOV     DX,03DAh                ;Point DX to CGA status
CAGetNext:
        CLI                             ;No interrupts now
CAWaitNoH:
        IN      AL,DX                   ;Get 6845 status
        TEST    AL,8                    ;Check for vert. retrace
        JNZ     CAGo                    ;In progress? Go
        RCR     AL,1                    ;Wait for end of hor.
        JC      CAWaitNoH               ; retrace
CAWaitH:
        IN      AL,DX                   ;Get 6845 status again
        RCR     AL,1                    ;Wait for horizontal
        JNC     CAWaitH                 ; retrace
CAGo:
        MOV     AL,AH                   ;Move Attr back to AL...
        STOSB                           ; and then to screen
        STI                             ;Allow interrupts
        INC     DI                      ;Skip characters
        LOOP    CAGetNext               ;Look for next opportunity
        JMP     CAExit                  ;Done
CANoWait:
        STOSB                           ;Change the attribute
        INC     DI                      ;Skip characters
        LOOP    CANoWait                ;Get next character
CAExit:                                 ;Next instruction
        MOV     SP,BP                   ;Restore SP
        POP     BP                      ;Restore BP
        RET     8                       ;Remove params and return

ChangeAttribute ENDP

;*******************************************************

;procedure MoveFromScreen(var Source, Dest; Length:Integer);

;Move Length words from Source (video mem.) to Dest w/o snow

MFLength        EQU     WORD PTR [BP+6]
MFDest          EQU     DWORD PTR [BP+8]
MFSource        EQU     DWORD PTR [BP+12]

MoveFromScreen  PROC FAR

        PUSH    BP                      ;Save BP
        MOV     BP,SP                   ;Set up stack frame
        MOV     BX,DS                   ;Save DS in BX
        MOV     AL,RetraceMode          ;Grab before changing DS
        LES     DI,MFDest               ;ES:DI points to Dest
        LDS     SI,MFSource             ;DS:SI points to Source
        MOV     CX,MFLength             ;CX = Length
        CLD                             ;Set direction to forward
        RCR     AL,1                    ;Check RetraceMode
        JNC     MFNoWait                ;False? Use MFNoWait
        MOV     DX,03DAh                ;Point DX to CGA status
```

```
MFNext:
        CLI                             ;No interrupts now
MFWaitNoH:
        IN      AL,DX                   ;Get 6845 status
        TEST    AL,8                    ;Check for vert. retrace
        JNZ     MFGo                    ;In progress? go
        RCR     AL,1                    ;Wait for end of hor.
        JC      MFWaitNoH               ; retrace
MFWaitH:
        IN      AL,DX                   ;Get 6845 status again
        RCR     AL,1                    ;Wait for horizontal
        JNC     MFWaitH                 ; retrace
MFGo:
        LODSW                           ;Load next vid. word to AX
        STI                             ;Allow interrupts
        STOSW                           ;Store video word in Dest
        LOOP    MFNext                  ;Get next video word
        JMP     MFExit                  ;All Done
MFNoWait:
        REP     MOVSW                   ;That's it!
MFExit:
        MOV     DS,BX                   ;Restore DS
        MOV     SP,BP                   ;Restore SP
        POP     BP                      ;Restore BP
        RET     10                      ;Remove params and return

MoveFromScreen  ENDP

;*******************************************************

;procedure MoveToScreen(var Source, Dest; Length : Integer);

;Move Length words from Source to Dest (vid. memory) w/o snow

MTLength        EQU     WORD PTR [BP+6]
MTDest          EQU     DWORD PTR [BP+8]
MTSource        EQU     DWORD PTR [BP+12]

MoveToScreen    PROC FAR

        PUSH    BP                      ;Save BP
        MOV     BP,SP                   ;Set up stack frame
        PUSH    DS                      ;Save DS
        MOV     AL,RetraceMode          ;Grab before changing DS
        LES     DI,MTDest               ;ES:DI points to Dest
        LDS     SI,MTSource             ;DS:SI points to Source
        MOV     CX,MTLength             ;CX = Length
        CLD                             ;Set direction to forward
        RCR     AL,1                    ;Check RetraceMode
        JNC     MTNoWait                ;False? Use MTNoWait
        MOV     DX,03DAh                ;Point DX to CGA status
MTGetNext:
        LODSW                           ;Load next vid. word to AX
        MOV     BX,AX                   ;Store video word in BX
        CLI                             ;No interrupts now
MTWaitNoH:
        IN      AL,DX                   ;Get 6845 status
        TEST    AL,8                    ;Check for vert. retrace
        JNZ     MTGo                    ;In progress? Go
        RCR     AL,1                    ;Wait for end of hor.
        JC      MTWaitNoH               ; retrace
MTWaitH:
        IN      AL,DX                   ;Get 6845 status again
        RCR     AL,1                    ;Wait for horizontal
        JNC     MTWaitH                 ; retrace
MTGo:
        MOV     AX,BX                   ;Move word back to AX...
        STOSW                           ; and then to screen
        STI                             ;Allow interrupts
        LOOP    MTGetNext               ;Get next video word
        JMP     MTExit                  ;All done
MTNoWait:
        REP     MOVSW                   ;That's all!
MTExit:
        POP     DS                      ;Restore DS
        MOV     SP,BP                   ;Restore SP
        POP     BP                      ;Restore BP
        RET     10                      ;Remove params and return

MoveToScreen    ENDP


CODE    ENDS

        END
```

```
unit SOOPGEN;
{ standard routines }

{$I COMPDIRS.PAS}

interface

uses Crt,Dos;

{$I SOOPDEFS.PAS}   { include program defaults & settings }


{============= GENERAL SERVICE ROUTINES ====================}

procedure Beep;
{ Puts 1/4 second of 440 Hz out on the speaker. }

function UserAbort:boolean;
{ allow user to abort an operation }

procedure Int24On;
{ enable new Int24 error handler }

procedure Int24Off;
{ restore original Int24 error handler }

procedure SetCursor(NewMode:byte);
{ Turns current cursor block/on/off }

function InsMode:byte;
{ determine if Insert is off (0) or on (1) }


{================ SCREEN HANDLING ROUTINES =================}

procedure FastWrite(St : string; Row, Col, Attr : Integer);
{ Writes St at Row,Col in Attr (video attribute) w/o snow }

procedure ChangeAttribute(Number, Row, Col, Attr : Integer);
{ Change Number vid. attributes to Attr starting at Row,Col }

procedure MoveToScreen(var Source, Dest; Length : Integer);
{ Moves Length words from Source to Dest w/o snow }

procedure MoveFromScreen(var Source, Dest; Length : Integer);
{ Moves Length words from Source to Dest without snow }

procedure WriteFast(X,Y,SC:byte; S:string);
{ write a string at X,Y in SC color }

procedure WriteAt(X,Y:byte; S:string);
{ write a string at X,Y with specified imbedded colors }

procedure WriteCaps(X,Y,C1,C2: byte; S:string);
{ write a string with Caps bolded }

procedure WriteVert(X,Y,Num,SC:byte; Ch:char);
{ repeat a character vertically Num times in SC color }

procedure WriteVertStr(X,Y,SC:byte; S:string);
{ repeat a string vertically in SC color }

procedure ScrollWindow(ULX,ULY,LRX,LRY,SC:byte;
                       Num:shortint);
{ scroll area of the screen Num lines & clear to SC }

procedure SaveWindow(ULX,ULY,LRX,LRY:byte;
                     var Saveto:WindowArray);
{ put screen in window storage for later recall }

procedure RestoreWindow(var RestFrom:WindowArray);
{ restore a previously saved window }

procedure SaveLines(StartLine,NumLines:byte; var Saveto);
{ save a number of 80 column screen lines }

procedure RestoreLines(StartLine,NumLines:byte;
                       var RestFrom);
{ restore a number of 80 column screen lines }

function GetScrChar(X,Y:byte):word;
{ get character and attribute from screen }

procedure ChangeScr(ULX,ULY,LRX,LRY,SC:byte);
{ change screen attribute of defined rectangle }


{============== STRING MANIPULATION ======================}

function CharStr(Ch:char; Len:byte):string;
{ return a string with Len chars }
```

```
function Value(S:String):real;
{ strips string S of blanks and converts to a real number }

procedure RoundIt(var R:real;Places:byte);
{ rounds R to Places accuracy }

function Equal(R1,R2:real):boolean;
{ check if 2 numbers are equal (removes rounding problem) }

function UpLowStr(S:String;CType:char):String;
{ convert string to upper/lower case }

function StripLeft(S:string; Ch:char):string;
{ strips Ch from left of S }

function StripRight(S:String; Ch:char):string;
{ strips Ch from right of S }

function PadLeft(S:string; Ch:char; Len:byte):string;
{ pads S with Ch on left to length Len }

function PadRight(S:string; Ch:char; Len:byte):string;
{ pads S with Ch on right to length Len }

function CenterStr(S:string; Ch:char; Len:byte):string;
{ center string S in field of Ch, Len characters wide }

function BytetoHex(V:byte):HexStr;
{ convert a byte to it's hex string equivalent }

function HextoByte(H:HexStr):byte;
{ convert a hex string to it's byte equivalent }

function MakeStr(var FData; M,N:integer; FType:char):string;
{ make a string from some type of data }

function FullPath(InPath:string; AddSlash:boolean):string;
{ build a full '\' delimited path string }

procedure MakeBox(ULX,ULY,LRX,LRY,SC,BType,Barline:byte;
                  Title:string);
{ draw a frame 0=no frame 1=single, 2=double, 3,4 w/title
  @Y+1 Barline puts horizontal line }
```

```
implementation


{============ GENERAL SERVICE ROUTINES ==================}

procedure Beep;
{ Puts 1/4 second of 440 Hz out on the speaker. }
begin
  if (not DefBeep) then Exit;
  Sound(440); Delay(125); Nosound;
end;

function UserAbort:boolean;
{ allow user to abort an operation }
var
  Ch   : byte;
  Regs : registers;
begin
  UserAbort:=False;
  if Keypressed then begin
    Regs.Ax:=$0000;   { read keyboard }
    Intr($16,Regs);
    if Lo(Regs.Ax)=$00 then Ch:=128+Hi(Regs.Ax)
    else Ch:=Lo(Regs.Ax);
    UserAbort:=(Ch=ESC);
  end;
  while Keypressed do begin  { clear keyboard buffer }
    Regs.Ax:=$0000;
    Intr($16,Regs);
  end;
end;

function IOResultPrim:Word;
{ Calls IoResult for Int24 }
begin
  IOResultPrim := IOResult;
end;

{$F+}
procedure
Int24(Flags,CS,IP,AX,BX,CX,DX,SI,DI,DS,ES,BP:word);interrupt;
{ general purpose critical error handler }
type
  ScrPtr = ^ScrBuf;
  ScrBuf = array [1..320] of byte;
```

```
var
  Display, OldLine: ScrPtr;
  AH,AL    : byte;
  OldAttr  : byte;
  Row,Col  : integer;
  Action   : char;
  ErrMsg   : string;
  ErrCode  : word;
  Ch       : shortint;
  DevAttr  : ^word;
  DevName  : ^char;
begin
  ErrCode:=IOResultPrim;   { call IOResult before to clear }
  if IsMono then Display:=ptr($B000,Pred(MSGLINE)*160)
  else Display:=ptr($B800,Pred(MSGLINE)*160);
  New(OldLine);
  OldLine^:=Display^;
  AH:=Hi(AX);
  AL:=Lo(AX);
  Col:=WhereX;
  Row:=WhereY;
  OldAttr:=TextAttr;
  ErrMsg:='';
  if (AH and $80) = 0 then begin
    ErrCode:=Lo(DI);
    case ErrCode of
      $00: ErrMsg:='Write protect';
      $01: ErrMsg:='Internal DOS';
      $02: ErrMsg:='Not ready';
      $03: ErrMsg:='Internal DOS';
      $04: ErrMsg:='Bad sector';
      $05: ErrMsg:='Internal DOS';
      $06: ErrMsg:='Seek';
      $07: ErrMsg:='Unknown media or bad disk';
      $08: ErrMsg:='Sector not found';
      $09: ErrMsg:='Printer out of paper';
      $0A: ErrMsg:='Write fault';
      $0B: ErrMsg:='Read fault';
      $0C: ErrMsg:='General failure';
      $0D: ErrMsg:='Bad FAT';
      else ErrMsg:='Unknown';
    end;
    if ErrCode<>$09 then ErrMsg:=ErrMsg+
      ' error on drive '+Chr(AL+65);
  end
  else begin
    DevAttr:= Ptr(BP, SI+4);  { point to device word }
    if (DevAttr^ and $8000) <> 0 then begin  { if bit 15 on }
      Ch:=0;
      repeat
        DevName:=Ptr(BP,SI+$0A+Ch);
        ErrMsg:=ErrMsg+DevName^;
        Inc(Ch);
      until (DevName^=Chr(0)) or (Ch>7);
      ErrMsg:=ErrMsg + ' not responding';
      ErrCode:=$02;
    end
    else begin
      ErrMsg:='Bad File Allocation Table';
      ErrCode:=$0D;
    end;
  end;
  GotoXY(1,MSGLINE);
  TextAttr:=ErrorC;
  ClrEol;
  Write(' ',ErrMsg,'  --  A)bort or R)etry?');
  Beep;
  repeat
    Action:=Upcase(Readkey);
  until Action in [#27,'A','R'];
  Display^:=OldLine^;
  Dispose(OldLine);
  GotoXY(Col,Row);
  TextAttr:=OldAttr;
  case Action of
    #27,'A': begin
           CritError:=ErrCode;
           AX:=0;
         end;
    'R': begin
           CritError:=0;
           AX:=1;
         end;
  end;
  ErrCode:=IOResultPrim;    { call IOResult after to clear }
end;
{$F-}
```

```
procedure Int24On;
{ enable new Int24 error handler }
begin
  GetIntVec($24,OldInt24);    { save old Int24 vector }
  SetIntVec($24,@Int24);      { install new error handler }
  CritError:=0;               {   and set global errors to 0 }
  PasError :=0;
  AMSTError:=0;
end;

procedure Int24Off;
{ restore original Int24 error handler }
begin
  SetIntVec($24,OldInt24);    { restore old Int24 vector }
end;

procedure SetCursor(NewMode:byte);
{ turns cursor block/on/off }
var
  Regs : Registers;
begin
  with Regs do begin
    AX := $0100;
    BX := $0000;
    CurrentCursor:=NewMode;
    case NewMode of
      0: CX:=CursorBlk;
      1: CX:=CursorOn;
      2: CX:=CursorOff;
    end;
  end;
  Intr($10,Regs);
end;

function InsMode:byte;
{ determine if Insert is off (0) or on (1) }
begin
  InsMode:= (Mem[$0000:$0417] and $80) shr 7;
end;
```

```
{=============== SCREEN HANDLING ROUTINES ==================}

{$L SOOPSCRN}          { load assembly language routines }
procedure FastWrite(St:string; Row,Col,Attr:Integer);
     external;
procedure ChangeAttribute(Number,Row,Col,Attr:Integer);
     external;
procedure MoveToScreen(var Source,Dest; Length:Integer);
     external;
procedure MoveFromScreen(var Source,Dest; Length:Integer);
     external;


procedure WriteFast(X,Y,SC:byte; S:string);
{ write a string at X,Y in SC color }
begin
  FastWrite(S,Y,X,SC);
end;

procedure WriteAt(X,Y:byte; S:string);
{ write a string at X,Y with specified imbedded colors }
var
  Attrs : array [0..6] of byte Absolute BackC;
  CAttr : byte;   { current attribute }
  Ps    : byte;   { current position }
  Len   : byte;   { length of string }
begin
  if Pos(#255,S)=0 then begin
    FastWrite(S,Y,X,NormC);
    Exit;
  end;
  CAttr:=NormC; { default to normal text }
  Ps:=0;
  Len:=Ord(S[0]);
  while Ps<Len do begin
    Inc(Ps);
    if S[Ps]=#255 then begin
      CAttr:=Attrs[Ord(S[Succ(Ps)])];
      Inc(Ps,2);
    end;
    FastWrite(S[Ps],Y,X,CAttr);
    Inc(X);
  end;
end;
```

```
procedure WriteCaps(X,Y,C1,C2: byte; S:string);
{ write a string in C1 with Caps in C2 }
var
  TStr  : array [1..160] of byte;
  Count : byte;
begin
  FillChar(TStr,160,C1);
  for Count:=1 to Length(S) do begin
    Move(S[Count],TStr[Pred(Count*2)],1);
    if S[Count] in ['A'..'Z',':'] then
      TStr[Succ(Pred(Count*2))]:=C2;
  end;
  MoveToScreen(TStr,Mem[ScreenAdr:(Pred(Y)*160+(Pred(X)*2))],
   Length(S));
end;

procedure WriteVert(X,Y,Num,SC:byte; Ch:Char);
{ repeat a character vertically Num times in SC color }
var
  Count : byte;
begin
  Count:=0;
  while Count<Num do begin
    FastWrite(Ch,Y+Count,X,SC);
    Inc(Count);
  end;
end;

procedure WriteVertStr(X,Y,SC:byte; S:string);
{ repeat a string vertically n SC color }
var
  Count : byte;
begin
  Count:=0;
  while Count<Length(S) do begin
    FastWrite(S[Succ(Count)],Y+Count,X,SC);
    Inc(Count);
  end;
end;
```

```
procedure ScrollWindow(ULX,ULY,LRX,LRY,SC:byte;
                       Num:shortint);
{ scroll an area of the screen Num lines & clear to SC }
var
  Regs   : registers;
  I,NC   : byte;
  Buffer : LineArray;
  BlankLine : string[80];
begin
  if ((not RetraceMode) or (Num=0)) then with Regs do begin
    if Num<0 then AH:=$06  { for scroll up }
    else AH:=$07;          { for scroll down }
    AL:=Abs(Num);          { scroll Num lines }
    CX:=Pred(ULY) shl 8 + Pred(ULX);
    DX:=Pred(LRY) shl 8 + Pred(LRX);
    BH:=SC;
    Intr($10,Regs);
  end
  else begin
    NC:=Succ(LRX-ULX);
    FillChar(BlankLine[1],NC,$20);
    BlankLine[0]:=Chr(NC);
    if Num>0 then begin
      for I := Pred(LRY) downto ULY do begin
        MovefromScreen(Mem[ScreenAdr:(Pred(I)*160+
          (Pred(ULX) shl 1))],Buffer,NC);
        MovetoScreen(Buffer,Mem[ScreenAdr:(I*160+
          (Pred(ULX) shl 1))],NC);
      end;
      FastWrite(BlankLine,ULY,ULX,SC);
    end
    else begin
      for I := ULY to Pred(LRY) do begin
        MovefromScreen(Mem[ScreenAdr:(I*160+
          (Pred(ULX) shl 1))],Buffer,NC);
        MovetoScreen(Buffer,Mem[ScreenAdr:(Pred(I)*160+
          (Pred( ULX) shl 1))],NC);
      end;
      FastWrite(BlankLine,LRY,ULX,SC);
    end;
  end;
end;
```

```
procedure SaveWindow(ULX,ULY,LRX,LRY:byte;
                     var SaveTo:WindowArray);
{ put screen in window storage for later recall }
var
  I,NC : byte;
begin
  NC:=Succ(LRX-ULX);
  for I := ULY to LRY do
    MoveFromScreen(Mem[ScreenAdr:(Pred(I)*160+
      (Pred(ULX) shl 1))],SaveTo.Add[Pred(I)],NC);
  SaveTo.ULX:=ULX;
  SaveTo.ULY:=ULY;
  SaveTo.LRX:=LRX;
  SaveTo.LRY:=LRY;
end;

procedure RestoreWindow(var RestFrom:WindowArray);
{ restore a previously saved window }
var
  I,NC : byte;
begin
  NC:=Succ(RestFrom.LRX-RestFrom.ULX);
  for I := RestFrom.ULY to RestFrom.LRY do
    MoveToScreen(RestFrom.Add[Pred(I)],Mem[ScreenAdr:
      (Pred(I) *160+(Pred(RestFrom.ULX) shl 1))],NC);
end;

procedure SaveLines(StartLine,NumLines:byte; var Saveto);
{ save a number of 80 column screen lines }
var
  I : byte;
begin
  for I:=StartLine to Pred(StartLine+NumLines) do
    MovefromScreen(Mem[ScreenAdr:Pred(I)*160],
      Mem[Seg(Saveto):Ofs(SaveTo)+((I-StartLine)*160)],80);
end;

procedure RestoreLines(StartLine,NumLines:byte;
                       var RestFrom);
{ restore a number of 80 column screen lines }
var  I : byte;
begin
  for I:=StartLine to Pred(StartLine+NumLines) do
    MoveToScreen(Mem[Seg(RestFrom):Ofs(RestFrom)+
      ((I-StartLin e)*160)],Mem[ScreenAdr:Pred(I)*160],80);
end;
```

```
function GetScrChar(X,Y:byte):word;
{ get character and attribute from screen }
var
  SCWord: Word;
begin
  MoveFromScreen(Mem[ScreenAdr:(Pred(Y)*160+
    (Pred(X) shl 1))],SCWord,1);
  GetScrChar:=SCWord;
end;

procedure ChangeScr(ULX,ULY,LRX,LRY,SC:byte);
{ change screen attribute of defined rectangle }
var
  Line,Cols : byte;
begin
  Cols:=Succ(LRX)-ULX;
  for Line:=ULY to LRY do
    ChangeAttribute(Cols, Line, ULX, SC);
end;


{=============== STRING MANIPULATION ======================}

function CharStr(Ch:char; Len:byte):string;
{ return a string with Len chars }
var
  S : string;
begin
  S[0] := Chr(Len);
  FillChar(S[1], Len, Ord(Ch));
  CharStr := S;
end;

function Value(S:string):real;
{ strips string S of blanks and converts to a real number }
var
  R    : real;
  Code : integer;
begin
  while Pos(' ',S)<>0 do Delete(S,Pos(' ',S),1);
  if S[Length(S)]='.' then S:=S+'0';
  Val(S,R,Code);
  Value:=R;
end;
```

```
procedure Roundit(var R:real;Places:byte);
{ rounds R to Places accuracy }
var
  S:String;
begin
  Str(R:12:Places,S);
  R:=Value(S);
end;

function Equal(R1,R2:real):boolean;
{ check if 2 numbers are equal (removes rounding problem) }
begin
  Equal:=(Abs(R1-R2)<0.0000001);
end;

function UpLowStr(S:String;CType:char):String;
{  convert string to upper/lower case }
var
  P:byte;
begin
  case CType of
    'L': for P:=1 to Length(S) do if S[P] in ['A'..'Z'] then
           S[P]:=Chr(Ord(S[P])+32);
    'U': for P:=1 to Length(S) do S[P]:=UpCase(S[P]);
  end;
  UpLowStr:=S;
end;

function StripLeft(S:string; Ch:char):string;
{ strips Ch from left of S }
var
  Done : boolean;
begin
  Done:=(Length(S)=0);
  while ((Length(S)>0) and (not Done)) do begin
    Done:=(Copy(S,1,1)<>Ch);
    if not Done then Delete(S,1,1);
  end;
  StripLeft:=S;
end;

function StripRight(S:string; Ch:char):string;
{ strips Ch from right of S }
var
  Done : boolean;
begin
  Done:=(Length(S)=0);
  while ((Length(S)>0) and (not Done)) do begin
    Done:=(Copy(S,Length(S),1)<>Ch);
    if not Done then Delete(S,Length(S),1);
  end;
  StripRight:=S;
end;

function PadLeft(S:string; Ch:char; Len:byte):string;
{ pads S with Ch on left to length Len }
begin
  while Length(S)<Len do S:=Ch+S;
  PadLeft:=S;
end;

function PadRight(S:string; Ch:char; Len:byte):string;
{ pads S with Ch on right to length Len }
begin
  while Length(S)<Len do S:=S+Ch;
  PadRight:=S;
end;

function CenterStr(S:string; Ch:char; Len:byte):string;
{ center string S in field of Ch, Len characters wide }
var
  TStr : string;
begin
  TStr:=StripLeft(StripRight(S,' '),' ');
  while Length(TStr)<Len do begin
    TStr:=TStr+Ch;
    if Length(TStr)<Len then TStr:=Ch+TStr;
  end;
  CenterStr:=TStr;
end;

function BytetoHex(V:byte):HexStr;
{ convert a byte to it's hex string equivalent }
const
  HEXCHARS : array [0..15] of char = '0123456789ABCDEF';
begin
  BytetoHex:=HEXCHARS[V div 16] + HEXCHARS[V mod 16];
end;
```

```
function HextoByte(H:HexStr):byte;
{ convert a hex string to it's byte equivalent }
const
  HEXCHARS : string[16] = '0123456789ABCDEF';
begin
  H:=PadLeft(H,'0',2);
  if ((Pos(H[1],HEXCHARS)>0) and (Pos(H[2],HEXCHARS)>0)) then
    HextoByte:=Pred(Pos(H[1],HEXCHARS)) shl 4 +
      Pred(Pos(H[2],HEXCHARS))
    else HextoByte:=0;
end;

function MakeStr(var FData; M,N:integer; FType:char):string;
{ make a string from some type of data }
var
  A   : string  absolute FData;
  B   : byte    absolute FData;
  C   : char    absolute FData;
  I   : integer absolute FData;
  L   : longint absolute FData;
  R   : real    absolute FData;
  W   : word    absolute FData;
  Y   : boolean absolute FData;
  TStr: string;
begin
  case FType of
    'A','E': TStr:=PadRight(A,' ',M);
    'B': if M>0 then Str(B:M,TStr) else Str(B,TStr);
    'C': begin
           TStr:=''+C;  TStr:=PadRight(TStr,' ',M);
         end;
    'H': if M>0 then TStr:=PadLeft(BytetoHex(B),' ',M)
         else TStr:=BytetoHex(B);
    'I': if M>0 then Str(I:M,TStr) else Str(I,TStr);
    'L': if M>0 then Str(L:M,TStr) else Str(L,TStr);
    'O': if Y then TStr:='On ' else TStr:='Off';
    'R': Str(R:M:N,TStr);
    'W': if M>0 then Str(W:M,TStr) else Str(W,TStr);
    'Y': begin
           if Y then TStr:='Y' else TStr:='N';
           TStr:=PadLeft(TStr,' ',M);
         end;
    else TStr:=CharStr(' ',M);
  end;
  MakeStr:=TStr;
end;
```

```
function FullPath(InPath:string; AddSlash:boolean):string;
{ build a full '\' delimited path string }
begin
  case AddSlash of
    True : if InPath[Length(InPath)]<>'\' then
             FullPath:=InPath+'\' else FullPath:=InPath;
    False: if Length(InPath)<3 then FullPath:=InPath+'\'
           else if Length(InPath)=3 then FullPath:=InPath
           else FullPath:=StripRight(InPath,'\');
  end;
end;

procedure MakeBox(ULX,ULY,LRX,LRY,SC,BType,Barline:byte;
                  Title:string);
{ draw a frame  0=no box, 1=single, 2=double, 3,4=w/title
  @Y+1 Barline puts line }
const
  WUL : array [1..2] of char = (#218,#201);
  WUR : array [1..2] of char = (#191,#187);
  WLL : array [1..2] of char = (#192,#200);
  WLR : array [1..2] of char = (#217,#188);
  WH  : array [1..2] of char = (#196,#205);
  WV  : array [1..2] of char = (#179,#186);
  WCL : array [1..2] of char = (#195,#199);
  WCR : array [1..2] of char = (#180,#182);
var
  I,NC : byte;
  TStr : string;
  LType: byte;
begin
  ScrollWindow(ULX,ULY,LRX,LRY,SC,0);
  if BType>0 then begin
    LType:=BType;
    if LType>2 then Dec(LType,2);
    NC:=Succ(LRX-ULX);
    TStr:=CharStr(WH[LType],NC);
    TStr[1]:=WUL[LType];
    TStr[NC]:=WUR[LType];
    FastWrite(TStr,ULY,ULX,SC);
    WriteVert(ULX,Succ(ULY),Pred(LRY-ULY),SC,WV[LType]);
    WriteVert(LRX,Succ(ULY),Pred(LRY-ULY),SC,WV[LType]);
    TStr[1]:=WLL[LType];
    TStr[NC]:=WLR[LType];
    FastWrite(TStr,LRY,ULX,SC);
    if Barline>0 then begin
      TStr:=CharStr(WH[1],NC);
```

```
    TStr[1]:=WCL[LType];
      TStr[NC]:=WCR[LType];
      FastWrite(TStr,ULY+Barline,ULX,SC);
    end;
  end;
  if Length(Title)>0 then case BType of
    0   : FastWrite(Title,ULY,ULX,SC);
    1,2 : FastWrite(' '+Title+' ',ULY,ULX+2,SC);
    3,4 : FastWrite(Title,Succ(ULY),ULX+2,SC);
  end;
end;



{============= PROGRAM STARTUP AND FINISH ==================}

{$F+} procedure AMSTExit;
{ gracefully exit program }
begin
  Release(AMSTTop);         { release heap space }
  TextAttr:=OrigTextAt;     { set text to original }
  ClrScr;                   { clear the screen }
  SetCursor(1);             { turn the cursor on }
  SetCBreak(DosBreakState);{ restore Dos Break State }
  Int24Off;
  ExitProc:=SavedExitProc; { restore original exit address }
end;
{$F-}


procedure SetProgramInit;
{ initialize program parameters }
var
  Regs  : registers;
  ScMode: byte;
  IsEga : boolean;
  I     : byte;
begin
  SavedExitProc:=ExitProc;   { save original exit address }
  ExitProc:=@AMSTExit;       { set new exit address }
  GetCBreak(DosBreakState);  { save Dos Break State }
  SetCBreak(False);          { now turn it off }
  CheckBreak:=False;         { Ctrl-Break checks off }
  OrigTextAt:=TextAttr;      { save original screen colors }
  with Regs do begin         { get screen mode & set param }
    AX:=$0F00;
    Intr($10,Regs);
    ScMode:=AL;
    AH := $12;     { check if EGA installed and selected }
    BL := $10;
    CX := $FFFF;
    Intr($10, Regs);
    IsEga := (CX <> $FFFF);
  end;
  if ScMode = 7 then begin   { must be mono screen    }
    ScreenAdr := $B000;      { Address of mono screen }
    CursorOn  := $0C0D;      { default cursor for mono }
    CursorBlk := $080D;      { default mono block cursor }
    BackC     := $00;        { background color }
    LowC      := $07;        { low text color   }
    NormC     := $0F;        { normal text color}
    InvC      := $70;        { inverse color    }
    HeadC     := $0F;        { headline color   }
    ErrorC    := $0F;        { error color      }
    HelpC     := $70;        { help line color  }
  end
  else ScreenAdr := $B800;   { otherwise is color monitor }
  RetraceMode := (ScMode<>7) and not(IsEga);  { snow check }
  IsMono:=(ScMode=7);        { is this a mono monitor? }
  CheckSnow:=RetraceMode;    { set Turbo's retrace check }
  TextAttr:=NormC;           { set current program color }
  Mark(AMSTTop);             { setup top of heap pointer }
  if MaxAvail>SizeOf(WindowArray) then New(OldScreen)
  else begin
    FastWrite(
     'Insufficient memory to run program....press any key',
      12,10,ErrorC);
    if Readkey<>#0 then ;  Halt;
  end;
  Int24On;                   { enable new error handler }
  ScrollWindow(1,1,80,25,NormC,0); { clear the screen }
  SetCursor(2);              { turn off the cursor }
  if InsMode=0 then Mem[$0000:$0417]:=Mem[$0000:$0417]+$80;
  Nosound;                   { make sure speaker is off }
  FillChar(CmdNum[1],MaxComList,1);   { current command }
  CmdList:=1;                { current command list }
  CurrCommand:=0;            { current command }
end;

begin
  SetProgramInit;
end.
```

```
unit SOOPGEN1;

{$I COMPDIRS.PAS}

interface

uses Crt,Dos,SOOPGEN;

{=============== USER INPUT ROUTINES ======================}

function Getakey:byte;
{ get a keystroke and do UserTask while waiting }

procedure WriteHelp(S:string);
{ show S on HELPLINE of screen in HelpC color }

procedure WriteMsg(SC:byte; S:string);
{ show S on MSGLINE line of screen in SC color }

procedure Msg(S:String);
{ put S on screen and wait for keypress }

function GetBool(S:String):boolean;
{ put S on screen and wait a Y/N/Esc answer }

function ErrorCheck(ShowMsg:boolean):boolean;
{ check if there was an I/O or critical error }

procedure ErrorMsg(ErrNum:byte);
{ show error message }

function HilightCommand(Option:integer):boolean;
{ hilight command by number or first letter (0 hilites 1st) }

procedure RunCommand(Selection:byte);
{ hilight correct command and set command number }

function FileExist(FileName:string):boolean;
{ returns True if file exists, False if file does not exist }

function PrinterReady:boolean;
{ get printer status }

function WritePrt(S:string):boolean;
{ print and check for errors or user abort }

function GetListV(X,Y,NI,CV:integer):integer;
{ allow user to select from vertical list }

function MakeFileName(var TFile:Str12):boolean;
{ try to make a valid file name from string }

function KeyBoard(OkSet:MenuSet; Cursor:byte):byte;
{ gets a valid keystroke and optionally runs pop-ups }

procedure ShowMenu(Num:byte);
{ show a menu in menu portion of screen }

function DBGetWorkingBuffers(var B1,B2,B3:DBBufPtr):boolean;
{ get buffers to store database records }

function DBMakeName(FName:string; FType,OptNum:byte):string;
{ make a filename }

procedure DBGetBuffer(var FData; ObjBuffer:DBBufPtr;
                      DFT:DBField);
{ get contents of buffer }

procedure DBPutBuffer(var FData; ObjBuffer:DBBufPtr;
                      DFT:DBField);
{ put contents into buffer }

procedure DBPutFieldDef(var DFT:DBField;
                        Title:DBTitleStr;FType:char;
                        Len,Decs,X,Y,Page,ALen:byte;
                        AOfs:integer; CCase:char;
                        Mand,Calc:boolean; KeyTyp:char;
                        OkSet:MenuSet; Form:DBFormStr;
                        WTitle:boolean);
{ put a definition into a DBField }

function DidMandatoryEntry(var FData; DBT:DBField):boolean;
{ check to see if this field has been entered }

procedure DBGetField(var FData; var Next:byte;DBT:DBField;
                     GType,SC:byte; ExRet:MenuSet);
{ field input  0=Update only  1=Update  2=Enter  3=Prompted }

function DBGetPrompted(var FData; Prompt:string; FType:char;
    X,Y,Len,Decs,Sc:byte; CCase:char; OkSet:MenuSet):boolean;
{ get prompted input from user }
```

```
procedure DBGetNextField(var FldNum:byte;Next:byte;
                         ObjF:DBFieldArray);
{ get the next field for data entry based on next pointer }

function DBLoadDef(FName:string;
                   ObjBuffer,ObjTBuffer,ObjBBuffer:DBBufPtr;
                   ObjF:DBFieldArray;
                   ObjScreen:WindowPtr):boolean;
{ load database definition }

procedure ObjectInit(ObjNum:byte; var ObjScreen:WindowPtr;
          var ObjBuffer,ObjTBuffer,ObjBBuffer:DBBufPtr;
          var ObjF:DBFieldArray);

function NextCompTime(Inst:InstType):real;
{ return the next time for a message > SimClock }

procedure SendMsg(FromCls:byte;FromInst:InstType; ToCls:byte;
                  ToInst:InstType; Message:MsgType;
                  Number,Clock:real);
{ send a Message }


implementation

{=============== USER INPUT ROUTINES ======================}


function Getakey:byte;
{ get a keystroke and do UserTask while waiting }
var
  Regs : registers;
begin
  repeat until KeyPressed;
  Regs.Ax:=$0000;       { read the keyboard }
  Intr($16,Regs);
  if Lo(Regs.Ax)=$00 then Getakey:=128+Hi(Regs.Ax)
    else Getakey:=Lo(Regs.Ax);
end;

procedure WriteHelp(S:string);
{ show S on HELPLINE of screen in HelpC color }
begin
  FastWrite(Padright(' '+S,' ',80),HELPLINE,1,HelpC);
end;
```

```
procedure WriteMsg(SC:byte; S:string);
{ show S on MSGLINE line of screen }
begin
  FastWrite(Padright(' '+S,' ',80),MSGLINE,1,SC);
end;

procedure Msg(S:String);
{ put S on screen and wait for keypress }
var
  OldCursor : byte;
  OldLine   : LineArray;
begin
  OldCursor:=CurrentCursor;     { save old cursor mode }
  SetCursor(2);                 { turn the cursor off }
  SaveLines(MSGLINE,1,OldLine); { save screen }
  WriteMsg(ErrorC,S+' (press any key)');
  Beep;
  if Getakey<>0 then;
  RestoreLines(MSGLINE,1,OldLine);     { restore screen }
  SetCursor(OldCursor);         { restore previous cursor }
end;

function GetBool(S:String):boolean;
{ put S on screen and wait a Y/N/Esc answer }
var
  OldCursor : byte;
  Ch        : byte;
  OldLine   : LineArray;
  Regs      : registers;
begin
  OldCursor:=CurrentCursor;     { save old cursor mode }
  SetCursor(2);                 { turn the cursor off }
  SaveLines(MSGLINE,1,OldLine); { save screen }
  WriteMsg(ErrorC,S+' (Y/N)');
  Beep;
  repeat                        { get the response }
    Ch:=Getakey;
    if not (Ch in YesNo+[ESC]) then Beep; { beep if invalid }
  until (Ch in YesNo+[ESC]);
  RestoreLines(MSGLINE,1,OldLine);    { restore screen }
  SetCursor(OldCursor);         { restore previous cursor }
  GetBool:=(Ch in Yes);
end;
```

```
function ErrorCheck(ShowMsg:boolean):boolean;
{ check if there was an I/O or critical error }
var
  IOENum : integer;
  IOMsg  : string;
begin
  IOENum:=IOResult;  { call IOResult to clear }
  if PASError<>0 then IOENum:=PASError;
  if IOENum=0 then IOENum:=PASError;
  case IOENum of
    $01: IOMsg:='Invalid function number';
    $02: IOMsg:='File not found';
    $03: IOMsg:='Path not found';
    $04: IOMsg:='Too many open files';
    $05: IOMsg:='Read-only or duplicate file, or non-empty
                 directory';
    $06: IOMsg:='Invalid file access code';
    $07: IOMsg:='Memory control blocks destroyed';
    $08: IOMsg:='Insufficient memory';
    $09: IOMsg:='Invalid memory block address';
    $0A: IOMsg:='Invalid environment';
    $0B: IOMsg:='Invalid format';
    $0C: IOMsg:='Invalid drive number';
    $0D: IOMsg:='Invalid data';
    $0E: IOMsg:='Unknown I/O error';
    $0F: IOMsg:='Invalid drive';
    else IOMsg:='Unknown I/O error';
  end;
  if ((CritError=0) and (IOENum>0)) then begin
    AMSTError:=(IOENum shl 8);
    if ShowMsg then Msg('Error: '+IOMsg);
  end
  else AMSTError:=CritError;
  CritError:=0;
  PASError :=0;
  ErrorCheck:=(AMSTError<>0);
end;

procedure ErrorMsg(ErrNum:byte);
{ show error message }
begin
  PASError:=ErrNum;
  if ErrorCheck(True) then ;
end;
```

```
function HilightCommand(Option:integer):boolean;
{ hilight the command by number or first letter }
var
  I      : byte;
  X,X1,Y : byte;
  LX     : byte;
  TStr   : string;
  MChr   : char;
begin
  TStr:=Commands.Line[1]+Commands.Line[2];  { work string }
  { get current X location }
  I:=0;
  LX:=Pos(':',TStr);
  Delete(TStr,1,LX);
  X:=1;
  while I<CmdNum[CmdList] do begin
    Inc(X);
    if TStr[X] in ['A'..'Z'] then Inc(I);
  end;
  MChr:=#0;
  case Option of
    0 : MChr:=TStr[X];
    1 : begin  { hilite next commnad }
          repeat
            if X=Length(TStr) then X:=1
            else Inc(X);
          until (TStr[X] in ['A'..'Z']);
          MChr:=TStr[X];
        end;
   -1: begin  { hilite previous command }
          repeat
            if X=1 then X:=Length(TStr)
            else Dec(X);
          until (TStr[X] in ['A'..'Z']);
          MChr:=TStr[X];
        end;
   else begin  { try to match a letter }
          X:=Pos(Upcase(Chr(Option)),TStr);
          if X>0 then MChr:=TStr[X];
        end;
  end; { case }
  { hilite the appropriate command and return command num }
  if MChr<>#0 then begin
    WriteCaps(1,MENULINE,LowC,NormC,' '+
      Commands.Line[1]+' ');
```

```
    WriteCaps(1,Succ(MENULINE),LowC,NormC,' '+
      Commands.Line[2]+' ');
    X:=LX;  Y:=MENULINE;
    TStr:=Commands.Line[1];
    CmdNum[CmdList]:=0;
    repeat
      Inc(X);
      if X=Length(TStr) then begin
        X:=1;  Y:=Succ(MENULINE);
        TStr:=Commands.Line[2];
      end;
      if (TStr[X] in ['A'..'Z']) then Inc(CmdNum[CmdList]);
    until (TStr[X]=MChr);
    X1:=X;
    while ((TStr[X1]<>' ') and (X1<Length(TStr))) do Inc(X1);
    if TStr[X1]<>' ' then Inc(X1);
    ChangeScr(X,Y,Succ(X1),Y,InvC);
    WriteHelp(Commands.Desc[CmdNum[CmdList]]);
  end;
  HilightCommand:=(MChr<>#0);
end;  { function HilightCommand }

procedure RunCommand(Selection:byte);
{ hilight correct command, and set done to true if new menu }
var
  OkCommand: boolean;
begin
  OkCommand:=False;
  case Selection of
    13  : OkCommand:=True;      { run current command }
    65..90,97..122 : OkCommand:=
          HilightCommand(Ord(Upcase(Chr(Selection))));
  end;
  if OkCommand then CurrCommand:=CmdNum[CmdList]
  else Beep;
end;

function FileExist(FileName:string):boolean;
{ returns True if file exists, False if file does not exist }
var
  SR : SearchRec;
begin
  FindFirst(FileName, ReadOnly + Hidden + SysFile, SR);
  FileExist:=(DosError=0) and (Pos('?',FileName)=0) and
             (Pos('*', FileName)=0);
end;
```

```
function PrinterReady:boolean;
{ get printer status }
var
  OldCursor : byte;
  OldLine   : LineArray;
  Test : boolean;
  Done : boolean;
  Ch   : byte;
  Regs : registers;
begin
  repeat
    Done:=False;
    Regs.Dx:=$0000;   { select printer 1 }
    Regs.Ax:=$0200;   { request printer status }
    Intr($17,Regs);
    Test:=((Hi(Regs.Ax) and 128)=128);  { printer ready? }
    if not Test then begin
      OldCursor:=CurrentCursor;     { save old cursor mode }
      SetCursor(2);                 { turn the cursor off }
      SaveLines(MSGLINE,1,OldLine);
      WriteMsg(ErrorC,'Printer not ready, A)bort R)etry?');
      Beep;
      repeat
        Ch:=GetaKey;
      until Ch in [ESC,65,82,97,114];
      Done:=Ch in [ESC,65,97];
      RestoreLines(MSGLINE,1,OldLine);
      SetCursor(OldCursor);
    end;
  until ((Done) or (Test));
  PrinterReady:=Test;
end;

function WritePrt(S:string):boolean;
{ print and check for errors or user abort }
const
  PWait   = 20000;  { 20 second wait for timeout }
var
  Regs    : registers;
  PAbort  : boolean;
  Chk     : byte;
  TimeOut : word;
begin
  if Length(S)=0 then begin
    WritePrt:=True;
    Exit;
```

```
  end;
  PAbort:=ErrorCheck(False);
  while ((Length(S)>0) and (not PAbort)) do begin
    if Keypressed then begin
      Regs.Ax:=$0000;          { read the keyboard }
      Intr($16,Regs);
      if Lo(Regs.Ax)=$00 then Chk:=128+Hi(Regs.Ax)
      else Chk:=Lo(Regs.Ax);
      if Chk=ESC then PAbort:=
        GetBool('Print cancel requested, Ok to stop?');
    end;
    if not PAbort then begin
      Regs.Dx:=$0000;          { select printer 1   }
      Regs.Ax:=Ord(S[1]);      { output 1 character }
      Intr($17,Regs);
      Timeout:=0;
      while (((Hi(Regs.Ax) and 128)=0) and
             (TimeOut<PWait)) do begin
        Inc(Timeout);  Delay(1);
        Regs.Dx:=$0000;   { select printer 1 }
        Regs.Ax:=$0200;   { request printer status }
        Intr($17,Regs);
      end;
      if Timeout=PWait then PAbort:=(not PrinterReady);
      if not PAbort then Delete(S,1,1);
    end;
  end;
  while Keypressed do begin
    Regs.Ax:=$0000;   { clear the keyboard, just in case }
    Intr($16,Regs);
  end;
  WritePrt:=(not PAbort);
end;

function GetListV(X,Y,NI,CV:integer):integer;
{ allow user to select from vertical list }
var
  I  : integer;
  LI : integer;  { longest item }
  CS : integer;
  OldCursor : byte;
  Ch,Ch1 : byte;
  OkSet : MenuSet;
  Dups : boolean;
  TStr : string[1];
  Title: string;
```

```
begin
  OldCursor:=CurrentCursor;    { save old cursor mode }
  SetCursor(2);                { turn the cursor off }
  LI:=0;  Dups:=False;
  for I:=0 to NI do if Length(VList[I])>LI then
    LI:=Length(VList[I]);
  Dups:=False;
  OkSet:=[];
  for I:=1 to NI do begin              { check for dups }
    TStr:=Copy(VList[I],1,1);  Ch1:=Ord(Upcase(TStr[1]));
    if (Ch1 in OkSet) then Dups:=True
    else begin
      OkSet:=OkSet+[Ch1];
      if Ch1 in [65..97] then OkSet:=OkSet+[Ch1+32];
    end;
  end;
  if Dups then OkSet:=[];
  SaveWindow(X,Y,X+LI+3,Y+NI+3,OldScreen^);
  Title:=CenterStr(VList[0],' ',LI);
  MakeBox(X,Y,X+LI+3,Y+NI+3,NormC,3,2,Title);
  for I:=1 to NI do FastWrite(VList[I],Y+2+I,X+2,LowC);
  if not Dups then ChangeScr(X+2,Y+3,X+2,Y+2+NI,NormC);
  CS:=CV;
  repeat                                  { get the response }
    ChangeScr(X+1,Y+2+CS,X+2+LI,Y+2+CS,InvC);
    repeat
      Ch:=GetaKey;
    until Ch in OkSet+[CR,ESC,UP,DOWN];
    ChangeScr(X+1,Y+2+CS,X+2+LI,Y+2+CS,LowC);
    if not Dups then ChangeScr(X+2,Y+2+CS,X+2,Y+2+CS,NormC);
    case Ch of
      32..126 : begin
                  CS:=1;
                  while UpLowStr(Copy(VList[CS],1,1),'U')<>
                    Upcase(Chr(Ch)) do Inc(CS);
                  Ch:=CR;
                end;
      UP  : if CS>1 then Dec(CS) else CS:=NI;
      DOWN: if CS<NI then Inc(CS) else CS:=1;
    end;
  until (Ch in [CR,ESC]);
  if Ch=ESC then CS:=0;
  RestoreWindow(OldScreen^);
  SetCursor(OldCursor);
  GetListV:=CS;
end;
```

```
function MakeFileName(var TFile:Str12):boolean;
{ try to make a valid file name from string }
var
  DotCount : integer;
  IsOk  : boolean;
  I     : integer;
begin
  IsOk:=True;
  TFile:=StripLeft(TFile,' ');
  if TFile<>'' then begin    { remove blanks }
    while Pos(' ',TFile)>0 do Delete(TFile,Pos(' ',TFile),1);
    DotCount:=0;             { count dots }
    for I:=1 to Length(TFile) do if TFile[I]='.' then
      Inc(DotCount);
    case DotCount of
      0 :  IsOk:=(Length(TFile)<9);
      1 :  IsOk:=((Pos('.',TFile)<10) and (Pos('.',TFile)>1)
             and ((Length(TFile)-Pos('.',TFile))<4));
      else IsOk:=False;
    end;
    TFile:=UpLowStr(TFile,'U');
  end
  else IsOk:=False;
  if ((not IsOk) and (TFile<>'')) then
    Msg('Illegal file name (must be XXXXXXXX.XXX form)');
  MakeFileName:=IsOk;
end;

function KeyBoard(OkSet:MenuSet; Cursor:byte):byte;
{ gets a valid keystroke and optionally runs pop-ups }
var
  Ch        : byte;
  OldCursor : byte;
  Regs      : registers;
begin
  OldCursor:=CurrentCursor;
  SetCursor(Cursor);
  repeat
    Ch:=Getakey;
    if not (Ch in OkSet) then Beep;  { beep if invalid }
  until (Ch in OkSet);
  KeyBoard:=Ch;
  SetCursor(OldCursor);
end;
```

```
procedure ShowMenu(Num:byte);
{ show a menu in menu portion of screen }

  procedure DoMenu1;  { Main Menu }
  begin
    case Paused of
      True  : Commands.Line[1] :=
                'MAIN MENU:  Clr  Delete  Enter  Load  Opt
                  Proceed  Report  Save  Update  Quit';
      False : Commands.Line[1] :=
                'MAIN MENU:  Clr  Delete  Enter  Load  Opt
                  Pause  Report  Save  Update  Quit';
    end;
    Commands.Line[2] :='            ';
    Commands.Desc[1] :=
      'CLR - Clear data from all objects in the simulation';
    Commands.Desc[2] :=
      'DELETE - Delete object instance from current class';
    Commands.Desc[3] :=
      'ENTER - Add new object instance to the current class';
    Commands.Desc[4] :=
      'LOAD - Load a simulation from disk or create new one';
    Commands.Desc[5] :='OPT - Miscellaneous program options';
    case Paused of
      True :Commands.Desc[6] :=
        'PROCEED - Proceed with the current simulation';
      False:Commands.Desc[6]:='PAUSE - Pause the simulation';
    end;
    Commands.Desc[7] :='REPORT - Print simulation reports';
    Commands.Desc[8] :='SAVE - Save simulation to disk';
    Commands.Desc[9] :='UPDATE - Update the current object';
    Commands.Desc[10]:='QUIT - Quit this program';
  end;

  procedure DoMenu126;  { record update }
  begin
    Commands.Line[1]:='  '+CLow+'          '+CNorm+
      '  '+CLow+'                '+CNorm+'F5'+CLow+
      '=Prev Record    '+CNorm+'  '+CLow+'                '+
      CNorm+'   '+CLow+'        ';
    Commands.Line[2]:='  '+CLow+'          '+CNorm+
      '  '+CLow+'                '+CNorm+'F6'+CLow+
      '=Next Record    '+CNorm+'F8'+CLow+'=Blank Field    '+
      CNorm+'F10'+CLow+'=Accept ';
  end;
```

```
  procedure DoMenu127;  { record entry }
  begin
    Commands.Line[1]:='  '+CLow+'          '+CNorm+'  '+
      CLow+'                  '+CNorm+'  '+CLow+
      '               '+CNorm+' '+CLow+'               '+
      CNorm+'   '+CLow+'        ';
    Commands.Line[2]:='  '+CLow+'          '+CNorm+' '+
      CLow+'                '+CNorm+'F6'+CLow+'=Next Record'+
      CNorm+'F8'+CLow+'=Blank Field  '+
      CNorm+'F10'+CLow+'=Accept ';
  end;

begin
  ScrollWindow(1,MENULINE,80,MENULINE+2,NormC,0);
  case Num of
    1  : DoMenu1;       { main menu }
    126: DoMenu126;     { update record }
    127: DoMenu127;     { enter record }
  end;
  if Num<100 then begin
    WriteCaps(1,23,LowC,NormC,' '+Commands.Line[1]+' ');
    WriteCaps(1,24,LowC,NormC,' '+Commands.Line[2]+' ');
    CmdList:=Num;
    CurrCommand:=0;
  end
  else begin
    WriteAt(2,MENULINE,Commands.Line[1]);
    WriteAt(2,Succ(MENULINE),Commands.Line[2]);
  end;
end;


function DBGetWorkingBuffers(var B1,B2,B3:DBBufPtr):boolean;
{ get buffers to store database records }
begin
  DBGetWorkingBuffers:=False;
  if MaxAvail<(3*SizeOf(DBBufArray))+MinMem then Exit;
  GetMem(B1,SizeOf(DBBufArray));
  GetMem(B2,SizeOf(DBBufArray));
  GetMem(B3,SizeOf(DBBufArray));
  DBGetWorkingBuffers:=True;
end;
```

```
function DBMakeKey(var FData; DFT:DBField):string;
{ make a key entry from some type of data }
var
  T   : string absolute FData;
  TStr: string;
  Ext : string;
  Ps  : byte;
begin
  case DFT.FType of
    'A','B','C','E','H','W','Y': begin
          TStr:=MakeStr(FData,DFT.Len,DFT.Decs,DFT.FType);
          TStr:=UpLowStr(TStr,'U');
          Ps:=Pos('-',TStr);
          if ((DFT.FType in ['I','L','R']) and
              (Ps>0)) then begin
            Delete(TStr,Ps,1);
            TStr:=#31+TStr;
          end;
        end;
    'I','L','R' : begin
          TStr:=MakeStr(FData,DFT.Len,DFT.Decs,DFT.FType);
          Ps:=Pos('-',TStr);
          if Ps>0 then begin
            Delete(TStr,Ps,1);
            TStr:=#31+
             PadLeft(StripLeft(TStr,' '),':',Pred(DFT.Len));
          end;
        end;
    else TStr:=CharStr(' ',DFT.Len);
  end;
  DBMakeKey:=TStr;
end;


function DBMakeName(FName:string; FType,OptNum:byte):string;
{ make a filename: 0-Def, 1-Class }
begin
  case FType of
    0: DBMakeName:=FName+'.DBD';
    1: DBMakeName:=FName+'.C'+BytetoHex(OptNum);
    else DBMakeName:=FName;
  end;
end;
```

```
procedure DBGetBuffer(var FData; ObjBuffer:DBBufPtr;
                      DFT:DBField);
{ get contents of buffer at defined field }
begin
  Move(ObjBuffer^[DFT.AOfs],FData,DFT.ALen);
end;

procedure DBPutBuffer(var FData; ObjBuffer:DBBufPtr;
                      DFT:DBField);
{ put contents into buffer }
begin
  Move(FData,ObjBuffer^[DFT.AOfs],DFT.ALen);
end;

procedure DBPutFieldDef(var DFT:DBField;
   Title:DBTitleStr;FType:char;
   Len,Decs,X,Y,Page,ALen:byte; AOfs:integer; CCase:char;
   Mand,Calc:boolean; KeyTyp:char; OkSet:MenuSet;
   Form:DBFormStr; WTitle:boolean);
{ put a definition into a DBField }
begin
  DFT.Title:=Title;   DFT.FType:=FType;
  DFT.Len:=Len;       DFT.Decs:=Decs;
  DFT.X:=X;           DFT.Y:=Y;
  DFT.Page:=Page;     DFT.ALen:=ALen;
  DFT.AOfs:=AOfs;     DFT.CCase:=CCase;
  DFT.Mand:=Mand;     DFT.Calc:=Calc;
  DFT.KType:=KeyTyp;  DFT.OkSet:=OkSet;
  DFT.Form:=Form;     DFT.WTitle:=WTitle;
end;

function DidMandatoryEntry(var FData; DBT:DBField):boolean;
{ check to see if this field has been entered }
var
  A : string  absolute FData;
  B : byte    absolute FData;
  C : char    absolute FData;
  I : integer absolute FData;
  R : real    absolute FData;
  W : word    absolute FData;
  Y : boolean absolute FData;
begin
  if not DBT.Mand then DidMandatoryEntry:=True
  else case DBT.FType of
    'A'    : DidMandatoryEntry:=(A<>CharStr(' ',DBT.Len));
    'B','H': DidMandatoryEntry:=(B<>DBBBYTE);
    'C'    : DidMandatoryEntry:=(C<>DBBCHAR);
    'E'    : DidMandatoryEntry:=(A<>DBBENTRY);
    'I'    : DidMandatoryEntry:=(I<>DBBINT);
    'R'    : DidMandatoryEntry:=(R<>DBBREAL);
    'W'    : DidMandatoryEntry:=(W<>DBBWORD);
  end;
end;

procedure DBGetField(var FData; var Next:byte;
                     DBT:DBField;GType,SC:byte;
                     ExRet:MenuSet);
{ field input 0=Update only  1=Update  2=Enter  3=Prompted }
var
  A : string  absolute FData;
  B : byte    absolute FData;
  C : char    absolute FData;
  I : integer absolute FData;
  R : real    absolute FData;
  W : word    absolute FData;
  Y : boolean absolute FData;
  TStr,TStr1: string;
  Ch,Nd     : byte;
  Ps,LPs,RPs,Mask: word;
  FKeys     : MenuSet;
  Formatted,
  ShowStar,
  NotEntered: boolean;
  OldR      : real;
  OldDate   : string[10];
  OldTime   : string[6];

  function Masked(Ps:word):boolean;
  begin
    Masked:=(((1 shl Pred(Ps)) and Mask) = (1 shl Pred(Ps)));
  end;

  procedure GetPs(Direction:integer);
  begin
    case Direction of
      -1 : begin  { previous position }
             Dec(Ps);
             if Formatted then
               while ((Masked(Ps)) and (Ps>0)) do Dec(Ps);
             if ((GType=3) and (Ps=0)) then begin
               Inc(Ps);
               while Masked(Ps) do Inc(Ps);
```

```
          end;
        end;
    1 : begin  { next position }
          Inc(Ps);
          if Formatted then
            while ((Masked(Ps)) and
              (Ps<=DBT.Len)) do Inc(Ps);
          if ((GType=3) and (Ps>DBT.Len)) then begin
            Dec(Ps);
            if Formatted then while Masked(Ps) do Dec(Ps);
          end;
        end;
  end;
end;


begin
  TStr:=MakeStr(A,DBT.Len,DBT.Decs,DBT.FType);
  if (DBT.FType in ['B','I','R','W']) then
    TStr:=PadRight(StripLeft(TStr,' '),' ',DBT.Len);
  WriteFast(DBT.X,DBT.Y,SC,TStr);
  Next:=CR;  { default }
  Mask:=0;
  Formatted:=(Mask<>0);
  ShowStar:=((Lo(GetScrChar((DBT.X-1),DBT.Y)) in [0,32,255])
    and (GType<>3));
  if ShowStar then
    WriteFast(Pred(DBT.X),DBT.Y,NormC+Blink,'*');
  Ps:=DBT.Len;
  if Formatted then while Masked(Ps) do Dec(Ps);
  RPs:=Ps;  Ps:=1;
  if Formatted then while Masked(Ps) do Inc(Ps);
  LPs:=Ps;
  FKeys:=[BACK,CR,ESC,LEFT,RIGHT,INSKEY,DELKEY,CTRLLEFT,
          CTRLR IGHT]+ExRet;
  case GType of
    0 : FKeys:=FKeys+[F8,F10,UP,PGUP,DOWN,PGDN];
    1 : FKeys:=FKeys+[F5,F6,F8,F10,UP,PGUP,DOWN,PGDN];
    2 : FKeys:=FKeys+[F6,F8,F10,UP,PGUP,DOWN,PGDN];
    4 : FKeys:=FKeys+[UP,DOWN];
  end;
  repeat
    if GType<3 then case InsMode of
      0 : WriteFast(70,MENULINE,InvC,'Insert Off');
      1 : WriteFast(70,MENULINE,InvC,'Insert On ');
    end;
    GotoXY(DBT.X+Pred(Ps),DBT.Y);
    Ch:=KeyBoard(DBT.OkSet+FKeys,1);
    if Ch in DBT.OkSet then begin
      if ((InsMode=0) or (Formatted) or
          (DBT.FType in ['B','H','I','R','W'])) then
        Delete(TStr,Ps,1)
      else Delete(TStr,Length(TStr),1);
      case DBT.CCase of
        'L': if Ch in [65..90] then Inc(Ch,32);
        'U': if Ch in [97..122] then Dec(Ch,32);
      end;
      Insert(Chr(Ch),TStr,Ps);
      WriteFast(DBT.X,DBT.Y,SC,TStr);
      GetPs(1);
    end
    else case Ch of
      BACK : if ((not Formatted) and (Ps>LPs)) then begin
               Dec(Ps);
               Delete(TStr,Ps,1);
               TStr:=TStr+' ';
               WriteFast(DBT.X,DBT.Y,SC,TStr);
             end;
      ESC,F3,F4,F5,F6,F7,F9,F10,UP,DOWN,PGUP,PGDN : Next:=Ch;
      F8   : begin           { blank field }
               case DBT.FType of
                 'A','N': TStr:=CharStr(' ',DBT.Len);
                 'B','I','W':
                    TStr:=PadRight('0',' ',DBT.Len);
                 'C'    : TStr:=DBBCHAR;
                 'E'    : TStr:=DBBENTRY;
                 'H'    : TStr:='00';
                 'R'    : TStr:=PadRight('0.0',' ',DBT.Len);
               end;
               WriteFast(DBT.X,DBT.Y,SC,TStr);
               Ps:=LPs;
             end;
      LEFT : begin           { left arrow }
               GetPs(-1);
               if Ps<LPs then Next:=LEFT;
             end;
      RIGHT: GetPs(1);       { right arrow }
      DELKEY:if not Formatted then begin  { Del }
               Delete(TStr,Ps,1);
               TStr:=TStr+' ';
               WriteFast(DBT.X,DBT.Y,SC,TStr);
             end;
```

```
    CTRLLEFT  : Ps:=LPs;          { start of line }
    CTRLRIGHT : if not Formatted then begin { end of line }
                  Ps:=DBT.Len;
                  while ((TStr[Ps]=' ') and (Ps>1)) do
                    Dec(Ps);
                  if ((TStr[Ps]<>' ') and (Ps<DBT.Len))
                    then Inc(Ps);
                end
                else Ps:=RPs;
  end;
until ((Ch in [CR,ESC,F1,F2,F3,F4,F5,F6,F7,F9,F10,
               UP,DOWN,PGUP,PGDN]) or
       (Ps<LPs) or (Ps>RPs));
if Next<>ESC then case DBT.FType of
  'A': A:=MakeStr(TStr,DBT.Len,DBT.Decs,DBT.FType);
  'B': begin
         OldR:=Value(TStr);
         if ((OldR<DBBMin) or (OldR>DBBMax)) then begin
           Msg('Value out of range (0..255)');
           Next:=NOKEY;
         end
         else B:=Lo(Round(OldR));
       end;
  'C': C:=TStr[1];
  'E': A:=PadLeft(StripLeft(TStr,' '),'0',DBT.Len);
  'H': B:=HextoByte(TStr);
  'I': begin
         OldR:=Value(TStr);
         if ((OldR<DBIMin) or (OldR>DBIMax)) then begin
           Msg('Value out of range (-32768..32767)');
           Next:=NOKEY;
         end
         else I:=Round(OldR);
       end;
  'R': begin
         OldR:=R;
         Ps:=0;
         Nd:=0;
         while ((Ps<Length(TStr)) and (Nd<2)) do begin
           Inc(Ps);
           if TStr[Ps]='.' then Inc(Nd);
           if Nd>1 then
             Delete(TStr,Ps,Length(TStr)-Pred(Ps));
         end;
         R := Value(TStr);
         RoundIt(R,DBT.Decs);
         TStr1:=MakeStr(R,DBT.Len,DBT.Decs,DBT.FType);
         if Length(TStr1)>DBT.Len then begin
           Msg('Value entered is out of acceptable range');
           Next:=NOKEY;
           R:=OldR;
         end;
       end;
  'W': begin
         OldR:=Value(TStr);
         if ((OldR<DBWMin) or (OldR>DBWMax)) then begin
           Msg('Value out of range (0..65535)');
           Next:=NOKEY;
         end
         else W:=Round(OldR);
       end;
  'Y': Y:=(TStr='Y');
end;
if (not (Next in [ESC,NOKEY])) then
  NotEntered:=(not DidMandatoryEntry(A,DBT))
else NotEntered:=False;
if NotEntered then begin
  Msg('Entry must be made here');
  Next:=NOKEY;
end;
if ShowStar then WriteFast(Pred(DBT.X),DBT.Y,LowC,' ');
WriteFast(DBT.X,DBT.Y,SC,
  MakeStr(FData,DBT.Len,DBT.Decs,DBT .FType));
end;


function DBGetPrompted(var FData; Prompt:string; FType:char;
           X,Y,Len,Decs,Sc:byte; CCase:char;
           OkSet:MenuSet):boolean;
{ get prompted input from user }
var
  OldS : array [0..2] of LineArray;
  OldI : byte;
  DFT  : DBField;
  SX   : byte;
  Next : byte;
  I    : byte;
begin
  SX:=X+Length(Prompt);
  DBPutFieldDef(DFT,' ',FType,Len,Decs,SX,Y,1,0,0,CCase,
    DBNMAND,DBNCALC,DBNKEY,OkSe t,DBBFORM,DBNTITLE);
```

```
  if ((Y in [1,25]) or (X=1)) then begin
    MoveFromScreen(Mem[ScreenAdr:Pred(Y)*160],OldS[0],80);
    MakeBox(X,Y,X+Length(Prompt)+Len,Y,NormC,0,0,Prompt);
  end
  else begin
    for I:=0 to 2 do
      MoveFromScreen(Mem[ScreenAdr:(Y-2+I)*160],OldS[I],80);
    MakeBox(X-2,Pred(Y),X+Length(Prompt)+Len+1,Succ(Y),
      NormC, 3,0,Prompt);
  end;
  OldI:=InsMode;
  if InsMode=1 then Mem[$0000:$0417]:=Mem[$0000:$0417]-$80;
  DBGetField(FData,Next,DFT,3,SC,EMPTYSET);
  if InsMode<>OldI then begin
    if InsMode=0 then Mem[$0000:$0417]:=Mem[$0000:$0417]+$80
    else Mem[$0000:$0417]:=Mem[$0000:$0417]-$80;
  end;
  if ((Y in [1,25]) or (X=1)) then
    MoveToScreen(OldS[0],Mem[ScreenAdr:Pred(Y)*160],80)
  else for I:=0 to 2 do
    MoveToScreen(OldS[I],Mem[ScreenAdr:(Y-2+I)*160],80);
  DBGetPrompted:=(Next<>ESC);
end;


procedure DBGetNextField(var FldNum:byte;Next:byte;
                         ObjF:DBFieldArray);
{ get the next field for data entry based on next pointer }
var
  Done: boolean;
  Direction : byte;
begin
  if Next=ESC then Exit;
  Done:=False;
  case Next of
    CR,DOWN: repeat
               if ObjF[FldNum]^.Page=0 then FldNum:=1
               else Inc(FldNum);
               Done:=((not ObjF[FldNum]^.Calc) and
                      (ObjF[FldNum]^.Page=1));
             until Done;
    LEFT,UP: repeat
               if FldNum=1 then
                 while ObjF[Succ(FldNum)]^.Page<>0 do
                   Inc(FldNum)
               else Dec(FldNum);
               Done:=((not ObjF[FldNum]^.Calc) and
                      (ObjF[FldNum]^.Page=1));
             until Done;
    PGDN,PGUP: ;
  end;
end;


function DBLoadDef(FName:string;
          ObjBuffer,ObjTBuffer,ObjBBuffer:DBBufPtr;
          ObjF:DBFieldArray; ObjScreen:WindowPtr):boolean;
{ load database definition }
var
  I   : byte;
  DBN : integer;
  SStr: string[DBMaxFldLen];
  DDFR: DBFileRec;
  DDFV: file of DBFileRec;
  NFlds : byte;
begin
  DBLoadDef:=False;
  if not FileExist(FName) then begin
    Msg('Database definition file '+FName+' not found ');
    Exit;
  end;
  NFlds:=0;
  FillChar(SStr,Succ(DBMaxFldLen),#32);
  ObjScreen^.ULX:=1;
  ObjScreen^.ULY:=Pred(DBMINY);
  ObjScreen^.LRX:=80;
  ObjScreen^.LRY:=Succ(DBMAXY);
  Assign(DDFV,FName);
  Reset(DDFV);
  if ErrorCheck(True) then Exit;
  FillChar(ObjBuffer^,Succ(DBMAXRECLEN),0);
  FillChar(ObjTBuffer^,Succ(DBMAXRECLEN),0);
  while not EOF(DDFV) do begin
    Read(DDFV,DDFR);
    if ErrorCheck(True) then begin
      Close(DDFV);
      Exit;
    end;
    case DDFR.RType of
      0: begin  { field definition }
           Inc(NFlds);
```

```
          Move(DDFR.FieldDef.Title,ObjF[NFlds]^,
               SizeOf(DBFie ld));
          ObjF[NFlds]^.Title:=
            PadRight(ObjF[NFlds]^.Title,' ',DBTITLELEN);
          case ObjF[NFlds]^.FType of
            'A': begin
                   SStr[0]:=Chr(ObjF[NFlds]^.Len);
                   DBPutBuffer(SStr,ObjBuffer,ObjF[NFlds]^);
                 end;
            'E': DBPutBuffer(DBBENTRY,ObjBuffer,
                   ObjF[NFlds]^);
          end;
        end;
      1: begin  { screen line }
          for I:=0 to 79 do case Hi(DDFR.ScrLine.Cont[I]) of
            1 : DDFR.ScrLine.Cont[I]:=
                  Lo(DDFR.ScrLine.Cont[I])+(LowC shl 8);
            2 : DDFR.ScrLine.Cont[I]:=
                  Lo(DDFR.ScrLine.Cont[I])+(NormC shl 8);
            3 : DDFR.ScrLine.Cont[I]:=
                  Lo(DDFR.ScrLine.Cont[I])+(InvC shl 8);
          end;
          Move(DDFR.ScrLine.Cont,
               ObjScreen^.Add[DDFR.ScrLine .Line],160);
        end;
    end;
  end;
  Close(DDFV);
  if ErrorCheck(True) then Exit;
  for I:=Succ(NFlds) to DBMAXFIELDS do ObjF[I]^:=ObjF[0]^;
  Move(ObjBuffer^,ObjBBuffer^,Succ(DBMAXRECLEN));
  DBLoadDef:=TRUE;
end;


procedure ObjectInit(ObjNum:byte; var ObjScreen:WindowPtr;
           var ObjBuffer,ObjTBuffer,ObjBBuffer:DBBufPtr;
                                 var ObjF:DBFieldArray);
var
  I  : integer;
  NFlds: byte;
  MemOk: boolean;
begin
  MemOk:=True;
  I:=0;
  if MaxAvail>SizeOf(WindowArray)+MinMem then
    GetMem(ObjScreen,SizeOf(WindowArray))
  else MemOk:=False;
  if MemOk then MemOk:=
    DBGetWorkingBuffers(ObjBuffer,ObjTBuffer,ObjBBuffer);
  if MemOk then begin
    I:=0;
    while ((I<=DBMAXFIELDS) and (MemOk)) do begin
      if MaxAvail>SizeOf(DBField)+MinMem then
        GetMem(ObjF[I],SizeOf(DBField))
      else MemOk:=False;
      Inc(I);
    end;
  end;
  if not MemOk then begin
    Msg('Insufficient memory to run program');
    Halt;
  end;
  with ObjF[0]^ do begin
    Title := CharStr(' ',10);
    FType := DBBCHAR;
    Len   := DBBBYTE;
    Decs  := DBBBYTE;
    X     := DBBBYTE;
    Y     := DBBBYTE;
    Page  := DBBBYTE;
    ALen  := DBBBYTE;
    AOfs  := DBBINT;
    CCase := DBUPLOW;
    Mand  := DBNMAND;
    Calc  := DBNCALC;
    KType := DBNKEY;
    OkSet := [];
    Form  := DBBFORM;
  end;
  if not DBLoadDef(DBMakeName(CLSNAMES[ObjNum],0,0),
    ObjBuffer,ObjTBuff er,ObjBBuffer,ObjF,ObjScreen) then
      Halt;
end;


function NextCompTime(Inst:InstType):real;
{ return the next time for a completion message > SimClock }
var
  TPtr : MsgPacketPtr;
begin
  NextCompTime:=SimClock+ROUNDFACT;
```

```
  TPtr:=FirstMsg;
  if TPtr=nil then Exit;
  while TPtr<>nil do begin
    if ((TPtr^.Clock>=SimClock) and
        (TPtr^.Message=SQ_COMPLETE) and
        (TPtr^.FromInst=Inst)) then begin
      NextCompTime:=TPtr^.Clock;
      Exit;
    end;
    TPtr:=TPtr^.Next;
  end;
end;


procedure SendMsg(FromCls:byte;FromInst:InstType; ToCls:byte;
                  ToInst:InstType; Message:MsgType;
                  Number,Clock:real);
{ send a Message }
var
  MsgPacket : MsgPacketPtr;
  TPtr      : MsgPacketPtr;
  LPtr      : MsgPacketPtr;
  Done      : boolean;
  MsgNum    : byte;
begin
  if MaxAvail<SizeOf(MsgPacketType)+MinMem then begin
    Msg('Insufficient memory for message queue');
    Halt;
  end;
  GetMem(MsgPacket,SizeOf(MsgPacketType));
  MsgPacket^.FromCls:=FromCls;
  MsgPacket^.FromInst:=FromInst;
  MsgPacket^.ToCls:=ToCls;
  MsgPacket^.ToInst:=ToInst;
  MsgPacket^.Message:=Message;
  MsgPacket^.Number:=Number;
  MsgPacket^.Clock:=Clock;
  if SStep then begin                    { display message }
    MsgNum:=Ord(MsgPacket^.Message);
    WriteMsg(NormC,
     'Send: '+ClsNames[MsgPacket^.FromCls]+
     ','+MsgPacket^.FromInst+
     ' to '+ClsNames[MsgPacket^.ToCls]+','+MsgPacket^.ToInst+
     ' "'+SoopMsgs[MsgNum]+'" '+
     MakeStr(MsgPacket^.Number,0,2,'R')+','+
     MakeStr(MsgPac ket^.Clock,0,2,'R'));
  if GetAKey<>0 then ;
  end;
  { determine where message fits in message queue }
  { the new message should be after equal clock time }
  Inc(MsgCount);
  WriteAt(60,1,CHead+MakeStr(MsgCount,5,0,'W')) ;
  MsgPacket^.Next:=FirstMsg;   { start as new first message }
  if FirstMsg=nil then begin   { this is the only message }
    FirstMsg:=MsgPacket;
    Exit;
  end;
  if MsgPacket^.Clock<FirstMsg^.Clock then begin  { first }
    MsgPacket^.Next:=FirstMsg;
    FirstMsg:=MsgPacket;
    Exit;
  end;
  Done:=False;
  TPtr:=FirstMsg;              { point to first message }
  LPtr:=TPtr;
  while (not Done) do begin    { find appropriate position }
    MsgPacket^.Next:=TPtr^.Next;
    TPtr^.Next:=MsgPacket;
    if TPtr<>FirstMsg then LPtr^.Next:=TPtr;
    Done:=(MsgPacket^.Next=nil);
    if not Done then begin
      LPtr:=TPtr;
      TPtr:=MsgPacket^.Next;
      Done:=(MsgPacket^.Clock<TPtr^.Clock);
    end;
  end;
end;


end.
```

```
unit SOOPENT;
{ Entity object unit }

{$I COMPDIRS.PAS}

interface

uses SOOPGEN,SOOPGEN1;

procedure EntClass(MsgPacket:MsgPacketType);
{ interface to the outside world }

implementation

const
  ObjNum = ENTITY;

type
  ObjRecPtr  = ^ObjRec;
  ObjRec     = record  { object record }
                 Status     : longint;
                 Instance   : InstType;
                 TypeCode   : real;
                 CurrLoc    : InstType;
                 CreateTime : real;
                 StartTime  : real;
                 TimeInSys  : real;
                 WillFail   : boolean;
                 Next       : ObjRecPtr;
                 Prev       : ObjRecPtr;
               end;

var
  FirstObj,CurrObj,LastObj,TPtr : ObjRecPtr;
  ObjScreen   : WindowPtr;     { object screen }
  ObjF        : DBFieldArray;  { field defs }
  ObjBuffer   : DBBufPtr;      { buffer for object }
  ObjBBuffer  : DBBufPtr;      { blank buffer for object }
  ObjTBuffer  : DBBufPtr;      { temp buffer for object }
  ObjSize     : word;          { size of this object }
  MData       : MsgPacketType; { working message }
  LastDisp    : pointer;       { last displayed object }
```

```
procedure ShowObject;
{ show current object }
var
  FData : DBFDataArray;
  FldNum: byte;
begin
  if CurrCls<>ObjNum then begin
    if (not SStep) then Exit;
    CurrCls:=ObjNum;
  end;
  if ((not SStep) and (CurrObj<>LastDisp) and (not Paused))
    then Exit;
  RestoreWindow(ObjScreen^);
  FldNum:=1;
  while ObjF[FldNum]^.Page=1 do begin
    DBGetBuffer(FData,ObjBuffer,ObjF[FldNum]^);
    with ObjF[FldNum]^ do
      WriteFast(X,Y,InvC,MakeStr(FData,Len,Decs,FType));
    Inc(FldNum);
  end;
  LastDisp:=CurrObj;
end;

procedure PutObjInBuffer;
{ put the Current object in the display buffer }
begin
  if CurrObj<>nil then Move(CurrObj^,ObjBuffer^,ObjSize)
  else Move(ObjBBuffer^,ObjBuffer^,ObjSize);
end;

procedure ClearCurrObject;
{ clear data from object }
var
  FData : DBFDataArray;
  FldNum: byte;
begin
  FldNum:=1;
  while ObjF[FldNum]^.Page=1 do begin
    if StripLeft(StripRight(ObjF[FldNum]^.Form,' '),'
      ')='BLANK' then begin
      DBGetBuffer(FData,ObjBBuffer,ObjF[FldNum]^);
      DBPutBuffer(FData,ObjBuffer,ObjF[FldNum]^);
    end;
    Inc(FldNum);
  end;
end;
```

```
function DeleteCurrObject:boolean;
begin
  DeleteCurrObject:=False;
  if CurrObj=nil then Exit;
  TPtr:=CurrObj;
  if FirstObj=TPtr then FirstObj:=FirstObj^.Next;
  if LastObj=TPtr then LastObj:=LastObj^.Prev;
  if CurrObj^.Prev<>nil then CurrObj:=CurrObj^.Prev
  else if CurrObj^.Next<>nil then CurrObj:=CurrObj^.Next
  else CurrObj:=nil;
  if TPtr^.Prev<>nil then TPtr^.Prev^.Next:=TPtr^.Next;
  if TPtr^.Next<>nil then TPtr^.Next^.Prev:=TPtr^.Prev;
  Dispose(TPtr);
  PutObjInBuffer;
  DeleteCurrObject:=True;
end;


function GetNewObject:boolean;
{ allocate a new object and add to end of linked list }
var
  TStr : InstType;
  TReal: real;
  Code : integer;
begin
  GetNewObject:=False;
  if MaxAvail<SizeOf(ObjRec)+MinMem then Exit;
  GetMem(TPtr,SizeOf(ObjRec));
  TPtr^.Prev:=LastObj;
  TPtr^.Next:=nil;
  if TPtr^.Prev<>nil then TPtr^.Prev^.Next:=TPtr;
  CurrObj:=TPtr;
  LastObj:=TPtr;
  if FirstObj=nil then FirstObj:=TPtr;
  Move(ObjBBuffer^,CurrObj^,ObjSize);
  if CurrObj^.Prev=nil then CurrObj^.Instance:='    1'
  else begin
    TReal:=Value(CurrObj^.Prev^.Instance)+1.0;
    CurrObj^.Instance:=MakeStr(TReal,5,0,'R');
  end;
  GetNewObject:=True;
end;
```

```
function GetNextObject:boolean;
{ get the next object }
begin
  GetNextObject:=False;
  if CurrObj=nil then Exit;
  if CurrObj^.Next=nil then Exit;
  CurrObj:=CurrObj^.Next;
  PutObjInBuffer;
  GetNextObject:=True;
end;


function GetPrevObject:boolean;
{ get the previous object }
begin
  GetPrevObject:=False;
  if CurrObj=nil then Exit;
  if CurrObj^.Prev=nil then Exit;
  CurrObj:=CurrObj^.Prev;
  PutObjInBuffer;
  GetPrevObject:=True;
end;


procedure ClearAllObjects;
{ clear all objects (note: this is special for entities) }
begin
  while DeleteCurrObject do;
end;


procedure LoadObjects;
{ load simulation objects from disk }
var
  TObj : ObjRec;
  ObF  : file of ObjRec;
begin
  { delete current objects from memory }
  while DeleteCurrObject do ;
  { read objects from disk file if the file exists }
  if not FileExist(DBMakeName(SimName,1,ObjNum)) then Exit;
  Assign(ObF,DBMakeName(SimName,1,ObjNum));
  Reset(ObF);
  while (not EOF(ObF)) do begin
    Read(ObF,TObj);
    if not GetNewObject then begin
```

```
      Close(ObF);
      Msg('Insufficient memory to load simulation');
      Halt;
    end;
    Move(TObj,CurrObj^,ObjSize);
  end;
  Close(ObF);
  CurrObj:=FirstObj;
  PutObjInBuffer;
  ShowObject;
end;


procedure SaveObjects;
{ save simulation objects to disk }
var
  ObF  : file of ObjRec;
begin
  { save objects to disk file }
  TPtr:=FirstObj;
  Assign(ObF,DBMakeName(SimName,1,ObjNum));
  Rewrite(ObF);
  while TPtr<>nil do begin
    Write(ObF,TPtr^);
    TPtr:=TPtr^.Next;
  end;
  Close(ObF);
end;


function PointTo(Inst:InstType):ObjRecPtr;
{ point to the indicated instance }
var
  TPtr : ObjRecPtr;
begin
  PointTo:=nil;
  if FirstObj=nil then Exit;
  TPtr:=FirstObj;
  while  TPtr<>nil do begin
    if TPtr^.Instance=Inst then begin
      PointTo:=TPtr;
      Exit;
    end;
    TPtr:=TPtr^.Next;
  end;
end;
```

```
procedure GenerateArrival;
{ generate an arrival of an entity }
begin
  { create a new entity }
  if not GetNewObject then Exit;
  { mark Instance id, arrival time, type code, status }
  CurrObj^.TypeCode:=MData.Number;
  CurrObj^.CreateTime:=MData.Clock;
  PutObjInBuffer;
  ShowObject;
  { request routing for self: "Where do I go?" }
  SendMsg(ENTITY,TPtr^.Instance,ROUTING,TPtr^.CurrLoc,
    GET_NEXT_RTE,TPtr^.TypeCode,SimClock);
  { generate next arrival of self }
  SendMsg(ENTITY,NINST,ROUTING,NINST,GEN_ARR_TIME,
    TPtr^.TypeC ode,SimClock);
end;


procedure RequestServQueGranted;
{ request for service/queue was granted, move entity to new
location }
begin
  TPtr:=PointTo(MData.ToInst);
  if TPtr=nil then Exit;

  { send message to prior location that entity is leaving }
  SendMsg(ENTITY,TPtr^.Instance,SERVQUE,TPtr^.CurrLoc,
    ENTITY_LEAVE_SQ,SimClock-TPtr^.StartTime,SimClock);

  { set service failure flag off }
  TPtr^.WillFail:=False;

  { set current location }
  TPtr^.CurrLoc:=MData.FromInst;

  { set current location start time }
  TPtr^.StartTime:=SimClock;
  CurrObj:=TPtr;
  PutObjInBuffer;
  ShowObject;
  { send return message to indicate that entity is moved and
    completion should be scheduled }
  SendMsg(ENTITY,TPtr^.Instance,ROUTING,TPtr^.CurrLoc,
    SCH_SQ_COMP,TPtr^.TypeCode,SimClock);
end;
```

```
procedure RequestServQueDenied;
{ request for service/queue was denied, attempt to
  reschedule/reroute }
begin
  TPtr:=PointTo(MData.ToInst);
  if TPtr=nil then Exit;
  CurrObj:=TPtr;
  PutObjInBuffer;
  ShowObject;
  if TPtr^.WillFail then begin  { entity destined to fail }
    SendMsg(ENTITY,TPtr^.Instance,ROUTING,TPtr^.CurrLoc,
     GET_FAIL_RTRY,TPtr^.TypeCode,SimClock);
  end
  else begin       { request alternate routing for entity }
    SendMsg(ENTITY,TPtr^.Instance,ROUTING,TPtr^.CurrLoc,
      GET_ALT_RTE,TPtr^.TypeCode,SimClock);
  end;
end;


procedure ServQueComplete;
{ service/queue completed, need next route }
begin
  TPtr:=PointTo(MData.ToInst);
  if TPtr=nil then Exit;
  CurrObj:=TPtr;
  PutObjInBuffer;
  ShowObject;
  if TPtr^.WillFail then begin  { entity destined to fail }
    SendMsg(ENTITY,TPtr^.Instance,ROUTING,TPtr^.CurrLoc,
      GET_FAIL_RTE,TPtr^.TypeCode,SimClock);
  end
  else begin  { send return message requesting next route }
    SendMsg(ENTITY,TPtr^.Instance,ROUTING,TPtr^.CurrLoc,
      GET_NEXT_RTE,TPtr^.TypeCode,SimClock);
  end;
end;

procedure SetFail(Fail:boolean);
{ set entity fail service flag }
begin
  TPtr:=PointTo(MData.ToInst);
  if TPtr=nil then Exit;
  TPtr^.WillFail:=Fail;
  CurrObj:=TPtr;  PutObjInBuffer;  ShowObject;
end;
```

```
procedure LeaveSystem;
{ entity leaves simulation }
begin
  TPtr:=PointTo(MData.ToInst);
  if TPtr=nil then Exit;
  { send message to prior location that entity is leaving }
  SendMsg(ENTITY,TPtr^.Instance,SERVQUE,TPtr^.CurrLoc,
    ENTITY_LEAVE_SQ,SimClock-TPtr^.StartTime,SimClock);
  TPtr^.CurrLoc:=NINST;
  TPtr^.TimeInSys:=SimClock-TPtr^.CreateTime;
  CurrObj:=TPtr;  PutObjInBuffer;  ShowObject;
  { send message indicating entity throughput }
  SendMsg(ENTITY,TPtr^.Instance,SIMULATE,NINST,ENTITY_DEP,
    TPtr^.TimeInSys,SimClock);
  { delete the entity, no longer needed }
  if DeleteCurrObject then;
end;

procedure EntClass(MsgPacket:MsgPacketType);
{ interface to the outside world }
begin
  MData:=MsgPacket;
  case MData.Message of
    CLEAR_OBJ     : ClearAllObjects;
    DELETE_OBJ    : if DeleteCurrObject then ShowObject;
    LOAD_OBJ      : LoadObjects;
    SAVE_OBJ      : SaveObjects;
    SHOW_CURR_OBJ : ShowObject;
    SHOW_NEXT_OBJ : if GetNextObject then ShowObject;
    SHOW_PREV_OBJ : if GetPrevObject then ShowObject;
    GEN_ARRIVAL   : GenerateArrival;
    REQ_SQ_GRANTED: RequestServQueGranted;
    REQ_SQ_DENIED : RequestServQueDenied;
    ENTITY_SQ_COMP: ServQueComplete;
    ENTITY_SET_FAIL:SetFail(True);
    ENTITY_NO_FAIL: SetFail(False);
    LEAVE_SYS     : LeaveSystem;
  end;
end;

begin
  ObjSize:=SizeOf(ObjRec)-8;   { subtract 8 for pointers }
  FirstObj:=nil; CurrObj:=nil; LastObj:=nil; LastDisp:=nil;
  ObjectInit(ObjNum,ObjScreen,ObjBuffer,ObjTBuffer,ObjBBuffer
,ObjF);
end.
```

```
unit SOOPRTE;
{ Routing object unit }

{$I COMPDIRS.PAS}

interface

uses SOOPGEN,SOOPGEN1;

procedure RteClass(MsgPacket:MsgPacketType);
{ interface to the outside world }

implementation

const
  ObjNum  = ROUTING;
const
  Dists:array [1..3] of InstType= ('UNFRM','EXPON','NORML');
type
  ObjRecPtr  = ^ObjRec;
  ObjRec     = record  { object record }
                 Status     : longint;
                 Instance   : InstType;
                 Desc       : string[25];
                 EntType    : real;
                 CurrLoc    : InstType;
                 Dist       : InstType;
                 Mean       : real;
                 Std        : real;
                 FailPerc   : real;
                 FailTo     : InstType;
                 NextLoc    : InstType;
                 BalkLoc    : InstType;
                 Next       : ObjRecPtr;
                 Prev       : ObjRecPtr;
               end;
var
  FirstObj,CurrObj,LastObj,TPtr : ObjRecPtr;
  ObjScreen   : WindowPtr;     { object screen }
  ObjF        : DBFieldArray;  { field defs }
  ObjBuffer   : DBBufPtr;      { buffer for object }
  ObjBBuffer  : DBBufPtr;      { blank buffer for object }
  ObjTBuffer  : DBBufPtr;      { temp buffer for object }
  ObjSize     : word;          { size of this object }
  MData       : MsgPacketType; { working message }
  LastDisp    : pointer;       { last displayed object }
```

```
procedure ShowObject;
{ show current object }
var
  FData : DBFDataArray;
  FldNum: byte;
begin
  if CurrCls<>ObjNum then begin
    if (not SStep) then Exit;
    CurrCls:=ObjNum;
  end;
  if ((not SStep) and (CurrObj<>LastDisp) and (not Paused))
    then Exit;
  RestoreWindow(ObjScreen^);
  FldNum:=1;
  while ObjF[FldNum]^.Page=1 do begin
    DBGetBuffer(FData,ObjBuffer,ObjF[FldNum]^);
    with ObjF[FldNum]^ do
      WriteFast(X,Y,InvC,MakeStr(FData,Len,Decs,FType));
    Inc(FldNum);
  end;
  LastDisp:=CurrObj;
end;

procedure PutObjInBuffer;
{ put the Current object in the display buffer }
begin
  if CurrObj<>nil then Move(CurrObj^,ObjBuffer^,ObjSize)
  else Move(ObjBBuffer^,ObjBuffer^,ObjSize);
end;

procedure ClearCurrObject;
{ clear data from object }
var
  FData : DBFDataArray;
  FldNum: byte;
begin
  FldNum:=1;
  while ObjF[FldNum]^.Page=1 do begin
    if StripLeft(StripRight(ObjF[FldNum]^.Form,' '),
      ' ')='BLANK' then begin
      DBGetBuffer(FData,ObjBBuffer,ObjF[FldNum]^);
      DBPutBuffer(FData,ObjBuffer,ObjF[FldNum]^);
    end;
    Inc(FldNum);
  end;
end;
```

```
function DeleteCurrObject:boolean;
begin
  DeleteCurrObject:=False;
  if CurrObj=nil then Exit;
  TPtr:=CurrObj;
  if FirstObj=TPtr then FirstObj:=FirstObj^.Next;
  if LastObj=TPtr then LastObj:=LastObj^.Prev;
  if CurrObj^.Prev<>nil then CurrObj:=CurrObj^.Prev
  else if CurrObj^.Next<>nil then CurrObj:=CurrObj^.Next
  else CurrObj:=nil;
  if TPtr^.Prev<>nil then TPtr^.Prev^.Next:=TPtr^.Next;
  if TPtr^.Next<>nil then TPtr^.Next^.Prev:=TPtr^.Prev;
  Dispose(TPtr);
  PutObjInBuffer;
  DeleteCurrObject:=True;
end;


function GetNewObject:boolean;
{ allocate a new object and add to end of linked list }
begin
  GetNewObject:=False;
  if MaxAvail<SizeOf(ObjRec)+MinMem then Exit;
  GetMem(TPtr,SizeOf(ObjRec));
  TPtr^.Prev:=LastObj;
  TPtr^.Next:=nil;
  if TPtr^.Prev<>nil then TPtr^.Prev^.Next:=TPtr;
  CurrObj:=TPtr;
  LastObj:=TPtr;
  if FirstObj=nil then FirstObj:=TPtr;
  Move(ObjBBuffer^,CurrObj^,ObjSize);
  GetNewObject:=True;
end;


function GetNextObject:boolean;
{ get the next object }
begin
  GetNextObject:=False;
  if CurrObj=nil then Exit;
  if CurrObj^.Next=nil then Exit;
  CurrObj:=CurrObj^.Next;
  PutObjInBuffer;
  GetNextObject:=True;
end;
```

```
function GetPrevObject:boolean;
{ get the previous object }
begin
  GetPrevObject:=False;
  if CurrObj=nil then Exit;
  if CurrObj^.Prev=nil then Exit;
  CurrObj:=CurrObj^.Prev;
  PutObjInBuffer;
  GetPrevObject:=True;
end;


procedure GetObject(RType:byte);
{ enter or update an object }
var
  FData  : DBFDataArray;
  Fin    : boolean;
  FldNum : byte;
  FFld   : byte;
  Next   : byte;
begin
  if ((RType=1) and (CurrObj=nil)) then Exit;
  Next:=CR;
  FldNum:=1;
  Fin:=True;
  while ObjF[FldNum]^.Calc do Inc(FldNum);
  FFld:=FldNum;
  ShowMenu(RType+125);
  repeat
    Fin:=False;
    Move(ObjBuffer^,ObjTBuffer^,ObjSize);
    if RType = 2 then begin
      Move(ObjBBuffer^,ObjBuffer^,ObjSize);
      if not GetNewObject then Next:=ESC;
    end;
    FldNum:=FFld;
    ShowObject;
    if Next<>ESC then repeat
      DBGetBuffer(FData,ObjBuffer,ObjF[FldNum]^);
      DBGetField(FData,Next,ObjF[FldNum]^,RType,InvC,
        EMPTYSET );
      DBPutBuffer(FData,ObjBuffer,ObjF[FldNum]^);
      DBGetNextField(FldNum,Next,ObjF);
    until Next in [ESC,F5,F6,F10];
    Fin:=(Next in [ESC,F10]);
```

```
    case Next of
      ESC: begin  { abort }
             Move(ObjTBuffer^,ObjBuffer^,ObjSize);
             if RType=2 then if DeleteCurrObject then ;
           end;
      F5 : begin  { previous object }
             Move(ObjBuffer^,CurrObj^,ObjSize);
             if not GetPrevObject then ;
           end;
      F6 : begin  { next object }
             Move(ObjBuffer^,CurrObj^,ObjSize);
             if RType=1 then if not GetNextObject then ;
           end;
      F10: Move(ObjBuffer^,CurrObj^,ObjSize);
    end;
    ShowObject;
  until Fin;
  ShowMenu(CmdList);
  if HilightCommand(0) then ;
end;


procedure ClearAllObjects;
{ clear data from all objects }
begin
  TPtr:=FirstObj;
  while TPtr<>nil do begin
    ClearCurrObject;
    TPtr:=TPtr^.Next;
  end;
end;


procedure LoadObjects;
{ load simulation objects from disk }
var
  TObj : ObjRec;
  ObF  : file of ObjRec;
begin
  { delete current objects from memory }
  while DeleteCurrObject do ;
  { read objects from disk file if the file exists }
  if not FileExist(DBMakeName(SimName,1,ObjNum)) then Exit;
  Assign(ObF,DBMakeName(SimName,1,ObjNum));
  Reset(ObF);
  while (not EOF(ObF)) do begin
    Read(ObF,TObj);
    if not GetNewObject then begin
      Close(ObF);
      Msg('Insufficient memory to load simulation'); Halt;
    end;
    Move(TObj,CurrObj^,ObjSize);
  end;
  Close(ObF);
  CurrObj:=FirstObj;  PutObjInBuffer;  ShowObject;
end;

procedure SaveObjects;
{ save simulation objects to disk }
var
  ObF  : file of ObjRec;
begin
  { save objects to disk file }
  TPtr:=FirstObj;
  Assign(ObF,DBMakeName(SimName,1,ObjNum));
  Rewrite(ObF);
  while TPtr<>nil do begin
    Write(ObF,TPtr^);
    TPtr:=TPtr^.Next;
  end;
  Close(ObF);
end;

function PointTo(CurrLoc:InstType; EntType:real):ObjRecPtr;
{ point to the routing record with the indicated current
  location and entity class }
var
  TPtr : ObjRecPtr;
begin
  PointTo:=nil;
  if FirstObj=nil then Exit;
  TPtr:=FirstObj;
  while  TPtr<>nil do begin
    if ((TPtr^.CurrLoc=CurrLoc) and ((TPtr^.EntType=EntType)
      or (EntType=0.0))) then begin
      PointTo:=TPtr;
      Exit;
    end;
    TPtr:=TPtr^.Next;
  end;
end;
```

```
function GetDistNumber(Dist:InstType; Mean,StdDev:real):real;
{ generate a number from the indicated distribution }
var
  DNum : byte;
  Found: boolean;

  function SampleNorm:real;
  { get a sample from a standard normal distribution }
  const
    { normal table sample (Schriber GPSS text p263) }
    NormTable1 : array [1..25] of real =
      (0.0,0.00003,0.00135,0.00621,0.02275,0.06681,0.11507,
       0.15866,0.21186,0.27425,0.34458,0.42074,0.5,0.57926,
       0.65542,0.72575,0.78814,0. 84134,0.88493,0.93319,
       0.97725,0.99379,0.99865,0.99997,1.0);
    NormTable2 : array [1..25] of real =
      (-5.0,-4.0,-3.0,-2.5,-2.0,-1.5,-1.2,-1.0,-0.8,-0.6,
       -0.4,-0.2,0.0,0.2,0.4,0.6,0.8,1.0,1.2,1.5,2.0,2.5,
        3.0,4.0,5.0);
  var
    RNum : real;
  begin
    RNum:=Random;   { get a random number }
    DNum:=1;
    while RNum>=NormTable1[Succ(DNum)] do begin
      Inc(DNum);
      if DNum=25 then begin
        SampleNorm:=NormTable1[DNum];
        Exit;
      end;
    end;
    SampleNorm:=NormTable1[DNum];
  end;

begin
  GetDistNumber:=0.0;
  DNum:=0;
  Found:=False;
  { find the correct distribution }
  while ((DNum<MaxDist) and (not Found)) do begin
    Inc(DNum);
    Found:=(Dists[DNum]=Dist);
  end;
  if not Found then Exit;
  case DNum of
    1 : begin { uniform (std dev is used as range) }
          GetDistNumber:= (Mean-StdDev)+(Random * StdDev*2);
        end;
    2 : begin { exponential (p163 of Schriber GPSS text) }
          GetDistNumber:= Mean * (-1.0 * ln(1-Random))
        end;
    3 : begin    { normal (see p262 of Schriber GPSS text) }
          GetDistNumber:= (StdDev * SampleNorm) + Mean;
        end;
  end;
end;


procedure GenerateArrivalTime;
{ determine which arrivals to generate & when (entity types
  if Number=0.0) }
var
  ATime : real;
  Found : boolean;
begin
  { find first route for the desired entity instance & type }
  TPtr:=PointTo(NINST,MData.Number);
  while TPtr<>nil do begin
    CurrObj:=TPtr;    { display the object for reference }
    PutObjInBuffer;
    ShowObject;
    { generate arrival time (offset by simulation clock) }
    ATime:=GetDistNumber(TPtr^.Dist,TPtr^.Mean,TPtr^.Std);
    { send message indicating entity should be generated }
    SendMsg(ROUTING,NINST,ENTITY,NINST,GEN_ARRIVAL,
      TPtr^.EntType,MData.Clock+ATime);
    { generate additional arrivals if desired }
    if MData.Number<>0.0 then Exit;
    repeat
      TPtr:=TPtr^.Next;
      if TPtr<>nil then Found:=(TPtr^.CurrLoc=NINST);
    until ((TPtr=nil) or (Found));
  end;
end;
```

```
procedure GetNextRoute(RouteCode:byte);
{ get next routing for entity and send appropriate messages }
{  RouteCode: 0 - Get primary next location                  }
{             1 - Get alternate route after primary denial  }
{             2 - Get fail route after failure of service   }
{             3 - Retry getting fail route after denial     }
begin
  { find the first route for desired entity instance & type }
  TPtr:=PointTo(MData.ToInst,MData.Number);
  if TPtr=nil then Exit;
  CurrObj:=TPtr;  PutObjInBuffer;  ShowObject;
  { if next location is blank, leave, else request entry }
  if (((TPtr^.NextLoc=NINST) and (RouteCode in [0,1])) or
      ((TPtr^.FailTo=NINST) and (RouteCode in [2,3]))) then
    SendMsg(ROUTING,NINST,ENTITY,MData.FromInst,LEAVE_SYS,
      0.0,SimClock)
  else case RouteCode of
    0 : begin  { no prior denials, try first location }
          if TPtr^.BalkLoc=NINST then
            SendMsg(ENTITY,MData.FromInst,SERVQUE,
              TPtr^.NextLoc,REQ_SQ_ENTRY,0.0,SimClock)
          else SendMsg(ENTITY,MData.FromInst,SERVQUE,
              TPtr^.NextLoc,REQ_SQ_ENTRY,1.0,SimClock)
        end;
    1 : if TPtr^.BalkLoc=NINST then begin
          { prior request failed, retry route with clock
            incremented to next completion time }
          SendMsg(ENTITY,MData.FromInst,SERVQUE,
            TPtr^.NextLoc,REQ_SQ_ENTRY,0.0,SimClock);
        end
        else begin  { request denied, try alternate route }
          SendMsg(ENTITY,MData.FromInst,SERVQUE,
            TPtr^.BalkLoc,REQ_SQ_ENTRY,0.0,SimClock);
        end;
    2 : begin       { service failed, request failure route }
          SendMsg(ENTITY,MData.FromInst,SERVQUE,TPtr^.FailTo,
            REQ_SQ_ENTRY,0.0,SimClock);
        end;
    3 : begin       { service failed, request failure route
                      repeated (reasoning like type 1) }
          SendMsg(ENTITY,MData.FromInst,SERVQUE,TPtr^.FailTo,
            REQ_SQ_ENTRY,0.0,SimClock);
        end;
  end;
end;
```

```
procedure ScheduleSrvQueCompletion;
{ schedule service/queue completion }
var  ATime : real;
begin
  TPtr:=PointTo(MData.ToInst,MData.Number);
  if TPtr=nil then Exit;
  CurrObj:=TPtr; PutObjInBuffer;
  ShowObject;
  ATime:=GetDistNumber(TPtr^.Dist,TPtr^.Mean,TPtr^.Std)+
    SimClock;
  if GetDistNumber('UNFRM',50.0,50.0) < TPtr^.FailPerc then
    SendMsg(ROUTING,NINST,ENTITY,MData.FromInst,
      ENTITY_SET_FAIL,0.0,ATime);
  SendMsg(ROUTING,MData.FromInst,SERVQUE,TPtr^.CurrLoc,
    SQ_COMPLETE,0.0,ATime);
end;

procedure RteClass(MsgPacket:MsgPacketType);
{ interface to the outside world }
begin
  MData:=MsgPacket;
  case MData.Message of
    CLEAR_OBJ     : ClearAllObjects;
    DELETE_OBJ    : if DeleteCurrObject then ShowObject;
    ENTER_OBJ     : GetObject(2);
    LOAD_OBJ      : LoadObjects;
    SAVE_OBJ      : SaveObjects;
    SHOW_CURR_OBJ : ShowObject;
    SHOW_NEXT_OBJ : if GetNextObject then ShowObject;
    SHOW_PREV_OBJ : if GetPrevObject then ShowObject;
    UPDATE_OBJ    : GetObject(1);
    GEN_ARR_TIME  : GenerateArrivalTime;
    GET_NEXT_RTE  : GetNextRoute(0);
    GET_ALT_RTE   : GetNextRoute(1);
    GET_FAIL_RTE  : GetNextRoute(2);
    GET_FAIL_RTRY : GetNextRoute(3);
    SCH_SQ_COMP   : ScheduleSrvQueCompletion;
  end;
end;

begin
  ObjSize:=SizeOf(ObjRec)-8;   { subtract 8 for pointers }
  FirstObj:=nil; CurrObj:=nil; LastObj:=nil; LastDisp:=nil;
  ObjectInit(ObjNum,ObjScreen,ObjBuffer,ObjTBuffer,
    ObjBBuffer,ObjF);
end.
```

```
unit SOOPSIM;
{ Simulation object unit }

{$I COMPDIRS.PAS}

interface

uses SOOPGEN,SOOPGEN1;

procedure SimClass(MsgPacket:MsgPacketType);
{ interface to the outside world }


implementation

const
  ObjNum = SIMULATE;

type
  ObjRecPtr  = ^ObjRec;
  ObjRec     = record  { object record }
                 Status     : longint;
                 Instance   : InstType;
                 Desc       : string[25];
                 MaxTime    : real;
                 CurrTime   : real;
                 CurrQty    : real;
                 MinTInSys  : real;
                 MaxTInSys  : real;
                 AvgTInSys  : real;
                 Next       : ObjRecPtr;
                 Prev       : ObjRecPtr;
               end;


var
  FirstObj,CurrObj,LastObj,TPtr : ObjRecPtr;
  ObjScreen   : WindowPtr;     { object screen }
  ObjF        : DBFieldArray;  { field defs }
  ObjBuffer   : DBBufPtr;      { buffer for object }
  ObjBBuffer  : DBBufPtr;      { blank buffer for object }
  ObjTBuffer  : DBBufPtr;      { temp buffer for object }
  ObjSize     : word;          { size of this object }
  MData       : MsgPacketType; { working message }
  LastDisp    : pointer;       { last displayed object }
```

```
procedure ShowObject;
{ show current object }
var
  FData : DBFDataArray;
  FldNum: byte;
begin
  if CurrCls<>ObjNum then begin
    if (not SStep) then Exit;
    CurrCls:=ObjNum;
  end;
  if ((not SStep) and (CurrObj<>LastDisp) and (not Paused))
    then Exit;
  RestoreWindow(ObjScreen^);  FldNum:=1;
  while ObjF[FldNum]^.Page=1 do begin
    DBGetBuffer(FData,ObjBuffer,ObjF[FldNum]^);
    with ObjF[FldNum]^ do
      WriteFast(X,Y,InvC,MakeStr(FData,Len,Decs,FType));
    Inc(FldNum);
  end;
  LastDisp:=CurrObj;
end;

procedure PutObjInBuffer;
{ put the Current object in the display buffer }
begin
  if CurrObj<>nil then Move(CurrObj^,ObjBuffer^,ObjSize)
  else Move(ObjBBuffer^,ObjBuffer^,ObjSize);
end;


procedure ClearCurrObject;
{ clear data from object }
var
  FData : DBFDataArray;
  FldNum: byte;
begin
  FldNum:=1;
  while ObjF[FldNum]^.Page=1 do begin
    if StripLeft(StripRight(ObjF[FldNum]^.Form,' '),
      ' ')='BLANK' then begin
      DBGetBuffer(FData,ObjBBuffer,ObjF[FldNum]^);
      DBPutBuffer(FData,ObjBuffer,ObjF[FldNum]^);
    end;
    Inc(FldNum);
  end;
end;
```

```
function DeleteCurrObject:boolean;
begin
  DeleteCurrObject:=False;
  if CurrObj=nil then Exit;
  TPtr:=CurrObj;
  if FirstObj=TPtr then FirstObj:=FirstObj^.Next;
  if LastObj=TPtr then LastObj:=LastObj^.Prev;
  if CurrObj^.Prev<>nil then CurrObj:=CurrObj^.Prev
  else if CurrObj^.Next<>nil then CurrObj:=CurrObj^.Next
  else CurrObj:=nil;
  if TPtr^.Prev<>nil then TPtr^.Prev^.Next:=TPtr^.Next;
  if TPtr^.Next<>nil then TPtr^.Next^.Prev:=TPtr^.Prev;
  Dispose(TPtr);
  PutObjInBuffer;
  DeleteCurrObject:=True;
end;


function GetNewObject:boolean;
{ allocate a new object and add to end of linked list }
begin
  GetNewObject:=False;
  if MaxAvail<SizeOf(ObjRec)+MinMem then Exit;
  GetMem(TPtr,SizeOf(ObjRec));
  TPtr^.Prev:=LastObj;
  TPtr^.Next:=nil;
  if TPtr^.Prev<>nil then TPtr^.Prev^.Next:=TPtr;
  CurrObj:=TPtr;
  LastObj:=TPtr;
  if FirstObj=nil then FirstObj:=TPtr;
  Move(ObjBBuffer^,CurrObj^,ObjSize);
  GetNewObject:=True;
end;


function GetNextObject:boolean;
{ get the next object }
begin
  GetNextObject:=False;
  if CurrObj=nil then Exit;
  if CurrObj^.Next=nil then Exit;
  CurrObj:=CurrObj^.Next;
  PutObjInBuffer;
  GetNextObject:=True;
end;
```

```
function GetPrevObject:boolean;
{ get the previous object }
begin
  GetPrevObject:=False;
  if CurrObj=nil then Exit;
  if CurrObj^.Prev=nil then Exit;
  CurrObj:=CurrObj^.Prev;
  PutObjInBuffer;
  GetPrevObject:=True;
end;


procedure GetObject(RType:byte);
{ enter or update an object }
var
  FData  : DBFDataArray;
  Fin    : boolean;
  FldNum : byte;
  FFld   : byte;
  Next   : byte;
begin
  if ((RType=1) and (CurrObj=nil)) then Exit;
  Next:=CR;
  FldNum:=1;
  Fin:=True;
  while ObjF[FldNum]^.Calc do Inc(FldNum);
  FFld:=FldNum;
  ShowMenu(RType+125);
  repeat
    Fin:=False;
    Move(ObjBuffer^,ObjTBuffer^,ObjSize);
    if RType = 2 then begin
      Move(ObjBBuffer^,ObjBuffer^,ObjSize);
      if not GetNewObject then Next:=ESC;
    end;
    FldNum:=FFld;
    ShowObject;
    if Next<>ESC then repeat
      DBGetBuffer(FData,ObjBuffer,ObjF[FldNum]^);
      DBGetField(FData,Next,ObjF[FldNum]^,RType,InvC,
        EMPTYSET);
      DBPutBuffer(FData,ObjBuffer,ObjF[FldNum]^);
      DBGetNextField(FldNum,Next,ObjF);
    until Next in [ESC,F5,F6,F10];
    Fin:=(Next in [ESC,F10]);
```

```
    case Next of
      ESC: begin  { abort }
             Move(ObjTBuffer^,ObjBuffer^,ObjSize);
             if RType=2 then if DeleteCurrObject then ;
           end;
      F5 : begin  { previous object }
             Move(ObjBuffer^,CurrObj^,ObjSize);
             if not GetPrevObject then ;
           end;
      F6 : begin  { next object }
             Move(ObjBuffer^,CurrObj^,ObjSize);
             if RType=1 then if not GetNextObject then ;
           end;
      F10: Move(ObjBuffer^,CurrObj^,ObjSize);
    end;
    ShowObject;
  until Fin;
  ShowMenu(CmdList);
  if HilightCommand(0) then ;
end;


procedure ClearAllObjects;
{ clear data from all objects }
begin
  TPtr:=FirstObj;
  while TPtr<>nil do begin
    ClearCurrObject;
    TPtr:=TPtr^.Next;
  end;
end;


procedure LoadObjects;
{ load simulation objects from disk }
var
  TObj : ObjRec;
  ObF  : file of ObjRec;
begin
  { delete current objects from memory }
  while DeleteCurrObject do ;
  { read objects from disk file if the file exists }
  if not FileExist(DBMakeName(SimName,1,ObjNum)) then Exit;
  Assign(ObF,DBMakeName(SimName,1,ObjNum));
  Reset(ObF);
  while (not EOF(ObF)) do begin
    Read(ObF,TObj);
    if not GetNewObject then begin
      Close(ObF);
      Msg('Insufficient memory to load simulation');
      Halt;
    end;
    Move(TObj,CurrObj^,ObjSize);
  end;
  Close(ObF);
  CurrObj:=FirstObj;
  PutObjInBuffer;
  ShowObject;
end;


procedure SaveObjects;
{ save simulation objects to disk }
var
  ObF  : file of ObjRec;
begin
  { save objects to disk file }
  TPtr:=FirstObj;
  Assign(ObF,DBMakeName(SimName,1,ObjNum));
  Rewrite(ObF);
  while TPtr<>nil do begin
    Write(ObF,TPtr^);
    TPtr:=TPtr^.Next;
  end;
  Close(ObF);
end;


procedure UpdateClock;
{ update the simulation object clock }
begin
  CurrObj^.CurrTime:=MData.Number;
  PutObjInBuffer;
  ShowObject;
  if CurrObj^.CurrTime>CurrObj^.MaxTime then
    SendMsg(SIMULATE,CurrObj^.Instance,MAILMAN,NINST,
      END_SIMULATION,0.0,PRIORITY);
end;
```

```
procedure ReportSimulation;
{ print all object detail }
var
  FData : DBFDataArray;
  FldNum: byte;
begin
  CurrObj:=FirstObj;
  if CurrObj=nil then Exit;
  if not PrinterReady then Exit;
  while CurrObj<>nil do begin
    PutObjInBuffer;
    FldNum:=1;
    while ObjF[FldNum]^.Page=1 do begin
      DBGetBuffer(FData,ObjBuffer,ObjF[FldNum]^);
      with ObjF[FldNum]^ do
        if not WritePrt(Title+
          ': '+MakeStr(FData,Len,Decs,FType)+PCRLF+PCRLF)
         then Exit;
      Inc(FldNum);
    end;
    if not WritePrt(PFF) then Exit;
    CurrObj:=CurrObj^.Next;
  end;
end;


procedure EntityDeparted;
{ an entity has left the system }
begin
  { set throughput }
  CurrObj^.CurrQty:=CurrObj^.CurrQty+1.0;
  { set min time in system }
  if ((MData.Number>0.0) and
     ((MData.Number<CurrObj^.MinTInSys) or
      (CurrObj^.MinTInSys=0.0))) then
    CurrObj^.MinTInSys:=MData.Number;
  { set max time in system }
  if MData.Number>CurrObj^.MaxTInSys then
    CurrObj^.MaxTInSys:=MData.Number;
  { set avg time in system }
  CurrObj^.AvgTInSys:=((CurrObj^.AvgTInSys*
    (CurrObj^.CurrQty- 1.0)) +
     MData.Number)/CurrObj^.CurrQty;
  PutObjInBuffer;
  ShowObject;
end;
```

```
procedure SimClass(MsgPacket:MsgPacketType);
{ interface to the outside world }
begin
  MData:=MsgPacket;
  case MData.Message of
    CLEAR_OBJ     : ClearAllObjects;
    DELETE_OBJ    : if DeleteCurrObject then ShowObject;
    ENTER_OBJ     : GetObject(2);
    LOAD_OBJ      : LoadObjects;
    SAVE_OBJ      : SaveObjects;
    SHOW_CURR_OBJ : ShowObject;
    SHOW_NEXT_OBJ : if GetNextObject then ShowObject;
    SHOW_PREV_OBJ : if GetPrevObject then ShowObject;
    UPDATE_OBJ    : GetObject(1);
    UPDATE_CLOCK  : UpdateClock;
    REPORT_SIM    : ReportSimulation;
    ENTITY_DEP    : EntityDeparted;
  end;
end;


begin
  ObjSize:=SizeOf(ObjRec)-8;   { subtract 8 for pointers }
  FirstObj:=nil;
  CurrObj:=nil;
  LastObj:=nil;
  LastDisp:=nil;
  ObjectInit(ObjNum,ObjScreen,ObjBuffer,ObjTBuffer,
    ObjBBuffer ,ObjF);
end.
```

```
unit SOOPSRV;
{ Service object unit }

{$I COMPDIRS.PAS}

interface

uses SOOPGEN,SOOPGEN1;

procedure SrvClass(MsgPacket:MsgPacketType);
{ interface to the outside world }

implementation

const
  ObjNum = SERVQUE;
type
  ObjRecPtr  = ^ObjRec;
  ObjRec     = record  { service record }
                 Status     : longint;
                 Instance   : InstType;
                 Desc       : string[25];
                 Capacity   : real;
                 SrvStatus  : StatusType;
                 CurrQty    : real;
                 MaxQty     : real;
                 AvgQty     : real;
                 TotalQty   : real;
                 Utilized   : real;
                 MinTBA     : real;
                 MaxTBA     : real;
                 MeanTBA    : real;
                 MinTime    : real;
                 MaxTime    : real;
                 MeanTime   : real;
                 LastArrival: real;
                 Next,Prev  : ObjRecPtr;
               end;
var
  FirstObj,CurrObj,LastObj,TPtr : ObjRecPtr;
  ObjScreen   : WindowPtr;     { object screen }
  ObjF        : DBFieldArray;  { field defs }
  ObjBuffer,ObjBBuffer,ObjTBuffer : DBBufPtr; { buffers }
  ObjSize     : word;          { size of this object }
  MData       : MsgPacketType; { working message }
  LastDisp    : pointer;       { last displayed object }
```

```
procedure ShowObject;
{ show current object }
var
  FData : DBFDataArray;
  FldNum: byte;
begin
  if CurrCls<>ObjNum then begin
    if (not SStep) then Exit;
    CurrCls:=ObjNum;
  end;
  if ((not SStep) and (CurrObj<>LastDisp) and (not Paused))
    then Exit;
  RestoreWindow(ObjScreen^);
  FldNum:=1;
  while ObjF[FldNum]^.Page=1 do begin
    DBGetBuffer(FData,ObjBuffer,ObjF[FldNum]^);
    with ObjF[FldNum]^ do
      WriteFast(X,Y,InvC,MakeStr(FData,Len,Decs,FType));
    Inc(FldNum);
  end;
  LastDisp:=CurrObj;
end;

procedure PutObjInBuffer;
{ put the Current object in the display buffer }
begin
  if CurrObj<>nil then Move(CurrObj^,ObjBuffer^,ObjSize)
  else Move(ObjBBuffer^,ObjBuffer^,ObjSize);
end;

procedure ClearCurrObject;
{ clear data from object }
var
  FData : DBFDataArray;
  FldNum: byte;
begin
  FldNum:=1;
  while ObjF[FldNum]^.Page=1 do begin
    if StripLeft(StripRight(ObjF[FldNum]^.Form,' '),
      ' ')='BLANK' then begin
      DBGetBuffer(FData,ObjBBuffer,ObjF[FldNum]^);
      DBPutBuffer(FData,ObjBuffer,ObjF[FldNum]^);
    end;
    Inc(FldNum);
  end;
end;
```

```
function DeleteCurrObject:boolean;
begin
  DeleteCurrObject:=False;
  if CurrObj=nil then Exit;
  TPtr:=CurrObj;
  if FirstObj=TPtr then FirstObj:=FirstObj^.Next;
  if LastObj=TPtr then LastObj:=LastObj^.Prev;
  if CurrObj^.Prev<>nil then CurrObj:=CurrObj^.Prev
  else if CurrObj^.Next<>nil then CurrObj:=CurrObj^.Next
  else CurrObj:=nil;
  if TPtr^.Prev<>nil then TPtr^.Prev^.Next:=TPtr^.Next;
  if TPtr^.Next<>nil then TPtr^.Next^.Prev:=TPtr^.Prev;
  Dispose(TPtr);
  PutObjInBuffer;
  DeleteCurrObject:=True;
end;


function GetNewObject:boolean;
{ allocate a new object and add to end of linked list }
begin
  GetNewObject:=False;
  if MaxAvail<SizeOf(ObjRec)+MinMem then Exit;
  GetMem(TPtr,SizeOf(ObjRec));
  TPtr^.Prev:=LastObj;
  TPtr^.Next:=nil;
  if TPtr^.Prev<>nil then TPtr^.Prev^.Next:=TPtr;
  CurrObj:=TPtr;
  LastObj:=TPtr;
  if FirstObj=nil then FirstObj:=TPtr;
  Move(ObjBBuffer^,CurrObj^,ObjSize);
  GetNewObject:=True;
end;


function GetNextObject:boolean;
{ get the next object }
begin
  GetNextObject:=False;
  if CurrObj=nil then Exit;
  if CurrObj^.Next=nil then Exit;
  CurrObj:=CurrObj^.Next;
  PutObjInBuffer;
  GetNextObject:=True;
end;
```

```
function GetPrevObject:boolean;
{ get the previous object }
begin
  GetPrevObject:=False;
  if CurrObj=nil then Exit;
  if CurrObj^.Prev=nil then Exit;
  CurrObj:=CurrObj^.Prev;
  PutObjInBuffer;
  GetPrevObject:=True;
end;


procedure GetObject(RType:byte);
{ enter or update an object }
var
  FData  : DBFDataArray;
  Fin    : boolean;
  FldNum : byte;
  FFld   : byte;
  Next   : byte;
begin
  if ((RType=1) and (CurrObj=nil)) then Exit;
  Next:=CR;
  FldNum:=1;
  Fin:=True;
  while ObjF[FldNum]^.Calc do Inc(FldNum);
  FFld:=FldNum;
  ShowMenu(RType+125);
  repeat
    Fin:=False;
    Move(ObjBuffer^,ObjTBuffer^,ObjSize);
    if RType = 2 then begin
      Move(ObjBBuffer^,ObjBuffer^,ObjSize);
      if not GetNewObject then Next:=ESC;
    end;
    FldNum:=FFld;
    ShowObject;
    if Next<>ESC then repeat
      DBGetBuffer(FData,ObjBuffer,ObjF[FldNum]^);
      DBGetField(FData,Next,ObjF[FldNum]^,RType,InvC,
        EMPTYSET);
      DBPutBuffer(FData,ObjBuffer,ObjF[FldNum]^);
      DBGetNextField(FldNum,Next,ObjF);
    until Next in [ESC,F5,F6,F10];
    Fin:=(Next in [ESC,F10]);
```

```
    case Next of
      ESC: begin  { abort }
             Move(ObjTBuffer^,ObjBuffer^,ObjSize);
             if RType=2 then if DeleteCurrObject then ;
           end;
      F5 : begin  { previous object }
             Move(ObjBuffer^,CurrObj^,ObjSize);
             if not GetPrevObject then ;
           end;
      F6 : begin  { next object }
             Move(ObjBuffer^,CurrObj^,ObjSize);
             if RType=1 then if not GetNextObject then ;
           end;
      F10: Move(ObjBuffer^,CurrObj^,ObjSize);
    end;
    ShowObject;
  until Fin;
  ShowMenu(CmdList);
  if HilightCommand(0) then ;
end;


procedure ClearAllObjects;
{ clear data from all objects }
begin
  TPtr:=FirstObj;
  while TPtr<>nil do begin
    ClearCurrObject;
    TPtr:=TPtr^.Next;
  end;
end;


procedure LoadObjects;
{ load simulation objects from disk }
var
  TObj : ObjRec;
  ObF  : file of ObjRec;
begin
  { delete current objects from memory }
  while DeleteCurrObject do ;
  { read objects from disk file if the file exists }
  if not FileExist(DBMakeName(SimName,1,ObjNum)) then Exit;
  Assign(ObF,DBMakeName(SimName,1,ObjNum));
  Reset(ObF);
  while (not EOF(ObF)) do begin
    Read(ObF,TObj);
    if not GetNewObject then begin
      Close(ObF);
      Msg('Insufficient memory to load simulation');
      Halt;
    end;
    Move(TObj,CurrObj^,ObjSize);
  end;
  Close(ObF);
  CurrObj:=FirstObj;
  PutObjInBuffer;
  ShowObject;
end;


procedure SaveObjects;
{ save simulation objects to disk }
var
  ObF  : file of ObjRec;
begin
  { save objects to disk file }
  TPtr:=FirstObj;
  Assign(ObF,DBMakeName(SimName,1,ObjNum));
  Rewrite(ObF);
  while TPtr<>nil do begin
    Write(ObF,TPtr^);
    TPtr:=TPtr^.Next;
  end;
  Close(ObF);
end;


procedure ReportSimulation;
{ print all object detail }
var
  FData : DBFDataArray;
  FldNum: byte;
begin
  CurrObj:=FirstObj;
  if CurrObj=nil then Exit;
  if not PrinterReady then Exit;
  while CurrObj<>nil do begin
    PutObjInBuffer;
    FldNum:=1;
```

```
    while ObjF[FldNum]^.Page=1 do begin
      DBGetBuffer(FData,ObjBuffer,ObjF[FldNum]^);
      with ObjF[FldNum]^ do
        if not WritePrt(Title+
          ': '+MakeStr(FData,Len,Decs,FType)+PCRLF+PCRLF)
          then Exit;
      Inc(FldNum);
    end;
    if not WritePrt(PFF) then Exit;
    CurrObj:=CurrObj^.Next;
  end;
end;


function PointTo(Loc:InstType):ObjRecPtr;
{ point to the indicated instance }
var
  TPtr : ObjRecPtr;
begin
  PointTo:=nil;
  if FirstObj=nil then Exit;
  TPtr:=FirstObj;
  while  TPtr<>nil do begin
    if TPtr^.Instance=Loc then begin
      PointTo:=TPtr;
      Exit;
    end;
    TPtr:=TPtr^.Next;
  end;
  PointTo:=TPtr;
end;


procedure SrvQueCompletion;
{ service/queue completion }
begin
  TPtr:=PointTo(MData.ToInst);
  if TPtr=nil then Exit;
  CurrObj:=TPtr;
  PutObjInBuffer;
  ShowObject;
  { message to indicate completion and next route required }
  SendMsg(SERVQUE,TPtr^.Instance,ENTITY,MData.FromInst,
    ENTITY_SQ_COMP,0.0,SimClock);
end;
```

```
procedure RequestServiceQueueEntry;
{ an entity is requesting entry }
begin
  TPtr:=PointTo(MData.ToInst);
  if TPtr=nil then Exit;
  if TPtr^.CurrQty<TPtr^.Capacity then begin
    { set to busy status }
    TPtr^.SrvStatus:=BUSY;
    TPtr^.CurrQty:=TPtr^.CurrQty+1.0;
    TPtr^.TotalQty:=TPtr^.TotalQty+1.0;
    { check for max quantity }
    if TPtr^.CurrQty>TPtr^.MaxQty then
      TPtr^.MaxQty:=TPtr^.CurrQty;

    { check for min interarrival time }
    if (((SimClock-TPtr^.LastArrival)>0.0) and
      (((SimClock-TPtr^.LastArrival)<TPtr^.MinTBA) or
        (TPtr^.MinTBA=0.0)))
      then TPtr^.MinTBA:=(SimClock-TPtr^.LastArrival);

    { check for max interarrival time }
    if (SimClock-TPtr^.LastArrival)>TPtr^.MaxTBA then
      TPtr^.MaxTBA:=(SimClock-TPtr^.LastArrival);

    { set mean time between arrivals }
    TPtr^.MeanTBA:=((TPtr^.MeanTBA*(TPtr^.TotalQty-1.0))+
      (Sim Clock-TPtr^.LastArrival))/TPtr^.TotalQty;

    { set last arrival time }
    TPtr^.LastArrival:=SimClock;

    { send message indicating request was granted }
    SendMsg(SERVQUE,TPtr^.Instance,ENTITY,MData.FromInst,
      REQ_SQ_GRANTED,0.0,SimClock);
  end
  else begin  { send message indicating request denied }
    if MData.Number=0.0 then  { no alternate, retry current }
      SendMsg(SERVQUE,TPtr^.Instance,ENTITY,MData.FromInst,
        REQ_SQ_DENIED,0.0,NextCompTime(TPtr^.Instance))
    else                      { there is an alternate route }
      SendMsg(SERVQUE,TPtr^.Instance,ENTITY,MData.FromInst,
        REQ_SQ_DENIED,0.0,SimClock);
  end;
  CurrObj:=TPtr; PutObjInBuffer; ShowObject;
end;
```

```
procedure EntityLeaveSrvQue;
{ tell service/queue that entity is leaving }
begin
  TPtr:=PointTo(MData.ToInst);
  if TPtr=nil then Exit;
  TPtr^.AvgQty:=(((TPtr^.TotalQty-1.0)*TPtr^.AvgQty)+
    TPtr^.CurrQty)/TPtr^.TotalQty;
  TPtr^.Utilized:=TPtr^.AvgQty*100.0/TPtr^.Capacity;
  TPtr^.CurrQty:=TPtr^.CurrQty-1.0;
  if TPtr^.CurrQty<1.0 then TPtr^.SrvStatus:=IDLE;
  if ((MData.Number>0.0) and ((MData.Number<TPtr^.MinTime) or
    (TPtr^.MinTime=0.0))) then TPtr^.MinTime:=MData.Number;
  if MData.Number>TPtr^.MaxTime then
    TPtr^.MaxTime:=MData.Number;
  TPtr^.MeanTime:=((TPtr^.MeanTime*TPtr^.TotalQty)+
    MData.Number)/(TPtr^.TotalQty+1.0);
  CurrObj:=TPtr;  PutObjInBuffer;  ShowObject;
end;

procedure SrvClass(MsgPacket:MsgPacketType);
{ interface to the outside world }
begin
  MData:=MsgPacket;
  case MData.Message of
    CLEAR_OBJ     : ClearAllObjects;
    DELETE_OBJ    : if DeleteCurrObject then ShowObject;
    ENTER_OBJ     : GetObject(2);
    LOAD_OBJ      : LoadObjects;
    SAVE_OBJ      : SaveObjects;
    SHOW_CURR_OBJ : ShowObject;
    SHOW_NEXT_OBJ : if GetNextObject then ShowObject;
    SHOW_PREV_OBJ : if GetPrevObject then ShowObject;
    UPDATE_OBJ    : GetObject(1);
    REPORT_SIM    : ReportSimulation;
    REQ_SQ_ENTRY  : RequestServiceQueueEntry;
    SQ_COMPLETE   : SrvQueCompletion;
    ENTITY_LEAVE_SQ:EntityLeaveSrvQue;
  end;
end;

begin
  ObjSize:=SizeOf(ObjRec)-8;   { subtract 8 for pointers }
  FirstObj:=nil; CurrObj:=nil; LastObj:=nil; LastDisp:=nil;
  ObjectInit(ObjNum,ObjScreen,ObjBuffer,ObjTBuffer,
    ObjBBuffer,ObjF);
end.
```

```
unit SOOPMSG;
{ Message passing unit }

{$I COMPDIRS.PAS}

interface

uses Crt,
     SOOPGEN,SOOPGEN1,       { general routines }
     SOOPSIM,                { simulation object unit }
     SOOPENT,                { entity object unit }
     SOOPRTE,                { routing object unit }
     SOOPSRV;                { service/queue object unit }

procedure MessageHandler;
{ main program message handler }

implementation

procedure CheckMessages;
{ check the message queue for pending messages }
var
  MData : MsgPacketType; { avoid pointers to retain data }
  TPtr  : MsgPacketPtr;
  Done  : boolean;
  MsgNum: byte;
begin
  Done:=(FirstMsg=nil);
  while not Done do begin
    if Keypressed then Exit;    { allows user to interrupt }
    if ((FirstMsg^.Clock>PRIORITY) and (Paused)) then Exit;
    if SStep then begin                { display message }
      MsgNum:=Ord(FirstMsg^.Message);
      WriteMsg(NormC,'Recv: '+ClsNames[FirstMsg^.FromCls]+
        ','+FirstMsg^.FromInst+
        ' to '+ClsNames[FirstMsg^.ToCls]+
        ','+FirstMsg^.ToInst+' "'+SoopMsgs[MsgNum]+'" '+
        MakeStr(FirstMsg^.Number,0,2,'R')+','+
        MakeStr(FirstMsg^.Clock,0,2,'R'));
      if GetAKey=ESC then begin
        Paused:=True;
        ShowMenu(1);      { show the correct command list }
        if HilightCommand(0) then ;
        Exit;
      end;
    end;
```

```
    if FirstMsg^.Clock>SimClock then begin   { update clock }
      SimClock:=FirstMsg^.Clock;
      SendMsg(MAILMAN,NINST,SIMULATE,NINST,UPDATE_CLOCK,
        SimClock,PRIORITY);
    end;
    MData:=FirstMsg^; { get message from message queue }
    TPtr:=FirstMsg;   { delete the message & reset pointers }
    FirstMsg:=FirstMsg^.Next;
    Dispose(TPtr);
    Dec(MsgCount);
    WriteAt(60,1,CHead+MakeStr(MsgCount,5,0,'W' ));
    case MData.ToCls of { send message to appropriate place }
      MAILMAN  : case MData.Message of { message to mailman }
                   END_SIMULATION : begin  { end simulation }
                                      Msg('Simulation completed');
                                      Paused:=True;
                                      ShowMenu(1);
                                      if HilightCommand(0) then ;
                                    end;
                 end;
      SIMULATE : SimClass(MData);
      ENTITY   : EntClass(MData);
      ROUTING  : RteClass(MData);
      SERVQUE  : SrvClass(MData);
    end;
    Done:=(FirstMsg=nil);
    if not Done then Done:=(FirstMsg^.Clock>PRIORITY);
  end;
end;

procedure ShowSimName;
{ show simulation name }
begin
  WriteAt(73,1,CHead+SimName);
end;

procedure ClearMessages;
{ clear the message queue }
var
  TPtr : MsgPacketPtr;
begin
  while FirstMsg<>nil do begin
    TPtr:=FirstMsg; FirstMsg:=FirstMsg^.Next; Dispose(TPtr);
  end;
  MsgCount:=0;
end;
```

```
procedure SimulationClear;
{ clear the simulation data from all objects }
begin
  if not
  GetBool('Are you sure you want to clear the simulation?')
    then Exit;
  ClearMessages;  { clear the message queue }
  SimClock:=0.0;  { set to a new simulation }
  { send message to each object class to clear itself }
  for CurrCls:=MaxClasses downto 1 do
    SendMsg(MAILMAN,NINST,CurrCls,NINST,CLEAR_OBJ,0.0,
      PRIORITY);
  SendMsg(MAILMAN,NINST,CurrCls,NINST,SHOW_CURR_OBJ,0.0,
    PRIORITY);
end;

procedure SimulationLoad;
{ load a simulation from disk }
var
  TName  : string[8];
begin
  if not GetBool('Ok to replace current simulation?') then
    Exit;
  { get name of simulation to load }
  TName:=SimName;
  if not
    DBGetPrompted(TName,'Enter Simulation Name to Load: ',
      'A',20,12,8,0,InvC,'U',FILECHAR) then Exit;
  if StripRight(StripLeft(TName,' '),' ')='' then Exit;
  if not FileExist(DBMakeName(TName,1,1)) then
    if not GetBool('Simulation '+TName+
      ' not found, Create new simulation?') then Exit;
  { set current simulation name and display }
  SimName:=TName;  ShowSimName;
  ClearMessages;
  SimClock:=0.0;
  Paused:=True;
  ShowMenu(1);        { show the correct command list }
  if HilightCommand(0) then ;
  { send message to each object class to load simulation }
  for CurrCls:=MaxClasses downto 1 do
    SendMsg(MAILMAN,NINST,CurrCls,NINST,LOAD_OBJ,0.0,
      PRIORITY);
  SendMsg(MAILMAN,NINST,CurrCls,NINST,SHOW_CURR_OBJ,0.0,
    PRIORITY);
end;
```

```
procedure SimulationSave;
{ save a simulation to disk }
var  TName : string[8];
begin
  { ask for filename to save }
  TName:=SimName;
  if not DBGetPrompted(TName,
    'Enter Simulation Name to Save: ','A',20,12,8,0,
    InvC,'U',FILECHAR) then Exit;
  if StripRight(StripLeft(TName,' '),' ')='' then Exit;
  { if it exists, ask about replacement }
  if FileExist(DBMakeName(TName,1,1)) then
    if not GetBool('Simulation '+TName+
    ' already exists, Ok to replace?') then Exit;
  { set current simulation name and display }
  SimName:=TName;
  ShowSimName;
  { send message to each object class to save itself }
  for CurrCls:=MaxClasses downto 1 do
    SendMsg(MAILMAN,NINST,CurrCls,NINST,SAVE_OBJ,0.0,
      PRIORITY);
  SendMsg(MAILMAN,NINST,CurrCls,NINST,SHOW_CURR_OBJ,0.0,
    PRIORITY);
end;

procedure SimulationReport;
{ print simulation reports }
var  CNum : byte;
begin
  for CNum:=1 to MaxClasses do  { tell each class to report }
   SendMsg(MAILMAN,NINST,CNum,NINST,REPORT_SIM,0.0,PRIORITY);
end;

procedure SimulationStartStop;
{ start and stop the simulation }
begin
  Paused:=(not Paused);   { unpause the simulation }
  ShowMenu(1);            { show the correct command list }
  if HilightCommand(0) then ;
  if ((not Paused) and (SimClock=0.0)) then begin
    RandSeed:=1;    { set the random number seed }
    { generate first arrival of all entity types }
    SendMsg(MAILMAN,NINST,ROUTING,NINST,GEN_ARR_TIME,0.0,
      SimClock);
  end;
end;
```

```
procedure SimulationOptions;
{ allow user to change simulation options }
begin
  VList[0]:='SIMULATION OPTIONS';
  VList[1]:='Beeper Toggle';
  VList[2]:='Single Step Toggle';
  case GetListV(32,16,2,1) of
    1 : DefBeep := (not DefBeep);
    2 : SStep   := (not SStep);
  end;
end;


procedure MessageHandler;
{ main program message handler }
var
  Ch      : byte;     { working character variable }
  M       : longint;  { temporary memory check variable }
begin
  CurrCls :=1;        { initialize currently diplayed class }
  Paused  :=True;     { current simulation is paused }
  SStep   :=False;    { single step is off }
  MsgCount:=0;        { message count is zero }
  SimClock:=0.0;      { set to a new simulation }
  FirstMsg:=nil;      { clear the message queue }
  WriteAt(1,1 ,CHead+
    'SIMULATION WITH OBJECT-ORIENTED PROGRAMMING
     Msg     Count:         Sim:         ');
  SimName :='        ';        { no current simulation }
  ShowMenu(1);                 { display menu }
  if HilightCommand(0) then ;  { hilite command list }
  SendMsg(MAILMAN,NINST,CurrCls,NINST,SHOW_CURR_OBJ,0.0,
    PRIORITY); { show first class object }
  repeat                       { go into command loop }
    if CurrCommand>0 then begin { user command pending }
      case CurrCommand of
        1 : SimulationClear;    { clear data in objects }
        2 : SendMsg(MAILMAN,NINST,CurrCls,NINST,
              DELETE_OBJ,0.0,PRIORITY);  { Delete object }
        3 : SendMsg(MAILMAN,NINST,CurrCls,NINST,
              ENTER_OBJ,0.0,PRIORITY);   { Enter object }
        4 : SimulationLoad;     { Load simulation from disk }
        5 : SimulationOptions;  { set simulation options }
        6 : SimulationStartStop;{ Proceed or Pause }
        7 : SimulationReport;   { Print simulation reports }
        8 : SimulationSave;     { Save simulation to disk }
```

```
      9 : SendMsg(MAILMAN,NINST,CurrCls,NINST,
            UPDATE_OBJ,0.0,PRIORITY);  { Update object }
      10: if GetBool('Are you sure you want to quit?') then
            Halt;      { Quit program }
    end;
    CurrCommand:=0;
  end
  else if Keypressed then begin
    Ch:=KeyBoard(AllChar+[BACK,CR,ESC,LEFT,RIGHT,PGUP,PGDN,
         F5,F6,178],2);
    if Ch=ESC then Ch:=81;
    case Ch of
      178  : begin  { show memory }
               M:=MaxAvail;
               Msg('Memory: '+MakeStr(M,0,0,'L'));
             end;
      F5   : SendMsg(MAILMAN,NINST,CurrCls,NINST,
               SHOW_PREV_OBJ,0.0,PRIORITY);  { show prev }
      F6   : SendMsg(MAILMAN,NINST,CurrCls,NINST,
               SHOW_NEXT_OBJ,0.0,PRIORITY);  { show next }
      PGUP : begin   { show previous class and object }
               CurrCls:=Succ((CurrCls+MaxClasses-2) mod
                 MaxClasses);
               SendMsg(MAILMAN,NINST,CurrCls,NINST,
                 SHOW_CURR_OBJ,0.0,PRIORITY);
             end;
      PGDN : begin   { show next class and object }
               CurrCls:=Succ(CurrCls mod MaxClasses);
               SendMsg(MAILMAN,NINST,CurrCls,NINST,
                 SHOW_CURR_OBJ,0.0,PRIORITY);
             end;
      BACK,LEFT  : if HilightCommand(-1) then;
      SPACE,RIGHT: if HilightCommand(1) then;
      13,33..47,58..126: RunCommand(Ch);
    end; { case }
  end
  else CheckMessages;
until False; { never leave loop! Program quits by HALT }
end;


end.
```